RISC-Linz
Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria, Europe

Systolic Multiprecision Arithmetic

Tudor JEBELEAN

(March, 1994)

RISC-Linz Report Series No. 94-37

Supported by: Austrian Bundesministerium für Wissenschaft und Forschung: doctoral scholarship and project 613.523/3-27a/89 (Gröbner Bases); Austrian Forschungsförderungsfonds: project S5302-PHY (Parallel Symbolic Computation); POSSO project(Polynomial Systems Solving – ESPRIT III BRA 6846).

PhD Thesis

Systolic Multiprecision Arithmetic

Dissertation
zur Erlangung des akademischen Grades
"Doktor der technischen Wissenschaften"

Eingereicht von
Dipl.-Ing. Tudor Jebelean

März 1993

Erster Begutachter: O. Univ.-Prof. Dr. Bruno Buchberger
Zweiter Begutachter: O. Univ.-Prof. Dr. George Collins

Angefertigt am Forschungsinstitut für Symbolisches Rechnen
Technisch-Naturwissenschaftliche Fakultät
Johannes Kepler Universität Linz

Supported by:

Austrian Bundesministerium für Wissenschaft und Forschung:
doctoral scholarship and project 613.523/3-27a/89 (Gröbner Bases);
Austrian Forschungsförderungsfonds:
project S5302-PHY (Parallel Symbolic Computation);
POSSO project(Polynomial Systems Solving – ESPRIT III BRA 6846);

Eidesstattliche Erklärung

Ich versichere, daß ich die Dissertation selbständing verfaßt habe, andere als die angegebenen Quellen und Hilfsmittel nicht verwendet und mich auch sonst keiner unerlaubten Hilfe bedient habe.

Tudor Jebelean
Linz, September 20, 1994

Abstract

We study systolic algorithms for performing long integer and long rational arithmetic in computer algebra systems. Novel algorithms for *exact division* and *GCD computation* are introduced, which work *least–significant digits first*, thus being suitable for aggregation with other least–significant digits first algorithms for multiplication and addition in an integrated unit for operation with rational numbers. Experiments on **Maspar** (SIMD architecture) and **hardware** implementations using Field Programmable Logic show that a significant speed–up can be achieved.

Abstract

Wir studieren systolische Algorithmen für Berechnungen mit großen ganzen Zahlen (*long integer arithmetic*) und rationalen Zahlen innerhalb *Computer Algebra* systemen. Es werden neue Algorithmen für **exakte Division** und **Berechnung des größten gemeinsamen Teilers** entwickelt, welche nach dem Prinzip *least–significant digits first* arbeiten. Diese können dann mit anderen *least–significant digits first* Algorithmen für Multiplikation und Addition kombiniert werden, innerhalb einer integrierten Einheit zwecks Manipulationen rationaler Zahlen. Durch Experimente auf Maspar (SIMD architektur) und Hardware Impelmentierung (Field Programmable Logic) prüfen wir daß eine signifikante Performance–steierung erzielt werden kann.

Contents

List of Figures

Chapter 1

Introduction

The lack of *efficiency* of the classical so called "von Neumann" architecture is due to the low use of hardware in todays computers. The *systolic paradigm* promises a spectacular improvement.

However, designing systolic algorithms seems much more difficult than designing "classical" algorithms. We approach in this thesis the systolic design of a particular class of algorithms, which are needed in algebraic computations. These are the algorithms for computations with *long integers*, which are then aggregated in order to obtain algorithms for *long rationals*.

The research started with the study of sequential and parallel algorithms, then proceeded to modify, adapt and sometimes create new algorithms, and finally went all the way down to the implementation details.

All this work would not have been possible without the constant help and encouragement from my advisors and my colleagues, and without the wonderful research environment which the institute RISC is.

1.1 Prologue

Dear reader,

The next section will open the *scientific* part of this thesis, which according to some immutable rules must be quite formal and strict, leaving very little room, if at all, for speculations and day-dreaming.

Before entering this part, please allow me to express some personal opinions which can be hardly sustained by solid facts and accurate scientific results, being rather based on approximative knowledge and vague feelings. These opinions, however, still have their roots in the present reality, and, most importantly, it is these opinions and feelings which provided me with the deep motivation for carrying out the research in this thesis.

Namely, I would like to talk about computers. Today's computers and tomorrow's computers. And when I am saying tomorrow's computers I am not thinking of Pentium or Sixtium, Alpha-chip or Beta, not even Paragon or its similar successors, nor some super-hyper-vector-Cray-like machine which might be currently sitting on the design table of the engineers. All these are, and will be, excellent machines, but they are still "today's" computers, because they are built on the same principles of today.

Which are wrong.

They might have been right decades ago when first computers were built, but the evolution of hardware technology made them obsolete.

In order to see this, let us compare the *computer* with another "computing" device, which is, in fact, the best we know. Let us compare it to the human *brain*:

Size: The brain has about 10^{10} neurons, and if I am not too mistaken much by equating a neuron with the amount of circuitry needed to realize one byte of RAM, then we get 10 GigaBytes, which is quite in the range of present technology (just think what amount of hardware is in a Connection Machine 5 with 64,000 processors, each with several MegaBytes of memory).

So here we are roughly even.

Speed: The nervous signals travel in our body with a speed of about 10 m/s (Compare this with the light-speed of electricity!). The switching speed of a neuron is somewhere between 1/1000 and 1/100 of a second (Compare this with hundreds of MegaHertz in present circuit technology!).

Hence, not only are computers faster, but they are by the orders of millions faster than the brain.

Performance: Yesterday when walking on the street you suddenly met a friend. It took you between 1/10 and 1/4 of a second to recognize him. Now ask your computer to do this.

Or ask the most powerful computer you know, with the best programs there are, to look at two 2D images as your eyes give to your brain, to reconstruct the 3D scene, to realize it is a street, to differentiate houses from cars and cars from trees and trees from people, to pick out faces from this scene and to differentiate them, to select a particular face from the crowd, and to finally come up with the idea:

"Hey, this is John! He looks like he's got a parking ticket."

I will not try to argue here about tasks which are better suited for computers as opposed to those which are better suited for brains. For one thing, brains are stubbornly opposed to being *programmed*, which makes it difficult to test their behavior for various tasks. My point is that the *brain* can clearly solve the tasks for which it is *already programmed* much faster and better than what we could expect from today's computers. And with the same amount of "hardware". This means that the "hardware" of the brain is used with much more

efficiency.

I am not saying that we should build computers to work like the human brain. I am just saying: "Yes, our computers are quite good. But there is out there another computing device which works thousands (millions?) of times more efficiently. So, aren't we having some efficiency problem with our computers? Isn't there a way to organize this hardware in such a way that it can be used more efficiently?"

Let us look at how the hardware is organized in today's computers. First there is some (small!) part of it which is called the *central processing unit* (CPU). Then there is another part of it, called *memory*, which is usually quite huge (in fact, the bigger it is, the better is the computer – we think) sitting around this CPU. Now how does this work? Well, the CPU is working all the time, while the memory, as I said, is just sitting around. OK, it is not exactly doing nothing, because, from time to time, the CPU will put or get some information into / from the memory. That is, from time to time **4 bytes** out of those **40 millions** will do something.

All in all, if you think of it, *at each particular moment*, only a tiny part of the hardware is participating in the computation: 1/1,000 or 1/1,000,000, or less. It is like a company in which the boss is working day and night, while each of its 1,000 employees is working only when the boss is in his office.

This is not very efficient, we are not pleased with the results of this company, hence we say: "Hey, we need more employees!" (That is: add more memory). The results might improve a little, but, of course, the efficiency decreases even more!

Then we say: "Change that boss there!" (put a faster CPU). Well, similar result.

OK, this did not work, then we put a board of directors. One to bring info from the workers, one to process it, one to decide where to send the result, and one to put it there. And they will work in

parallel.

Rings a bell? Yes, this is the *RISC architecture*, which indeed increases the efficiency of our system, because now a larger part of the hardware is working all the time. So instead of 1/5,000 we have 1/1,000, which is a great improvement, but there is still room enough.

Which is the other architectural improvement now indispensable to any computer? It is *cache memory*, which also brings some parallelism in the use of the hardware, this time on the side of the memory. But again, just by a small amount.

So, IT'S PARALLELISM THAT WE NEED! Let us put ten such companies together, a hundred, a thousand! (That is *coarse grained parallelism.*) Now look at the results! Isn't it GREAT? Of course it is, because 10 horses pull better than one, just, this is not an automobile! The efficiency of this device cannot be higher than the efficiency of the components, and, in fact, it is even lower, because the directors of the companies must now communicate from time to time, and while they are conferring over the phone, what do their employees do? Yes, you are right, really NOTHING. Hence the efficiency decreases even further, sometimes by a large factor.

Because, indeed, we need parallelism, but *a lot of it.* The *brain* solves in **hundreds** of steps complex problems which require millions of operations because it works in parallel with thousands of tiny processors. It is like a company in which [a large part of] the employees work together. This is what we need for significantly improving the efficiency of the hardware usage in computers. Rather than having the hardware just sitting around, we should have it *working* around, that is, it should be organized in thousands of tiny processors, working together.

Well, that's clear to many, and has been for a long time. Hardware engineers can easily build this kind of thing for us. However we want it. The only problem is: **How do we want it?**.

And this is THE BIG PROBLEM! For the lack of solutions to which we do not have THE efficient computers. If you have thousands of processors, each with very limited capability, how do you interconnect them? How do you organize the computation such that all (most, many) of them work together? There has been a lot of research on these things, and quite soon people found out that solving these *organizational* problems is is often much more difficult than the initial work itself. (Which is quite natural: a company in which all the people work is more difficult to organize ...) This brings us to the heart of the matter.

We need a way to organize complex computations on a very large number of small processors.

And now I can finally say that:

My thesis is not going to solve this problem.

But, at least, I am going to attack it. And because so many people attacked it top-down, frontally, in fair-fight, and failed, I will attack it bottom-up, by the side, viciously at Achilles' heel. Namely, I will **not** try to find an *universally* perfect architecture and a way to efficiently map *any* algorithm on this architecture. Rather, I will consider a certain architecture,

the systolic one,

which is quite simple, and yet has the basic characteristics of what we are looking for, and I will try to implement efficiently on it a certain class of algorithms:

long integer arithmetic in symbolic computation,

which is limited, but large enough to be of practical interest.

This approach has the obvious disadvantage that it will not solve the problem (which is what is not expected from me anyway), but it also has some advantages:

- It leads to some practical algorithms / implementations which are useful for applications.

- It gives some insight into "Why things [do not] work" with this model, which is a [small] step forward towards solving the problem.
- It demonstrates that something useful *can* be done on this model, shattering a little the "good old" conviction that the sequential model is the best we can have.
- It does not end by giving the 397-th algorithm for parallel matrix multiplication, while still not solving the general problem.

There are some further reasons why I choose the systolic model and this particular area of applications, but explaining this already becomes "scientific", so please turn the page to see how the real story begins, and

enjoy!

1.2 Object of the thesis

The title "*Systolic multiprecision* `arithmetic`" indicates
the `problem` we want to solve,
the *domain* of the problem,
and the *method.*
The context of our approach is symbolic (or algebraic) computation.

The `problem` is arithmetic, that is we want to efficiently perform operations like **addition, multiplication, division, GCD computation**.

The *domain* is that of **long integers**, that is very big integer numbers which can only be represented using several words (e.g. of 32 bits) in a conventional computer. We are interested in the range from 1 up to 100 words, because this is what occurs in typical algebraic computations. For the same reasons, we are looking for algorithms which perform well on operands of *various* sizes, which is different, for instance, from what is needed in cryptography. The domain also contains **long rationals**, that is, rational numbers which are represented by pairs of long integers. Because rational operations are sequences of integral operations, one has to find integral algorithms which perform well *together*. This is particularly interesting in the case of systolic algorithms, because if partial results (digits) can be pipelined between the different stages of computation, then these stages can overlap in time, leading to a significant increase of performance.

As a "side effect", sometimes algorithms for operations with **polynomials** are developed, because they are similar to the algorithms for long integers, and also because they are the main "users" of integral and rational algorithms.

The *method* is **systolic parallelization.** The term systolic was introduced by [Kung and Leiserson, 1978] (see also [Kung and Leiserson, 1980, Kung, 1982]) and designates a parallel computation in which the data is "systolically" pumped from the memory through an array of processors and back again (see Fig. 1.1), exactly like the blood is pumped by the heart through the circulatory system.

The array could be also two- or three-dimensional, and the flow of information could be more intricate (different directions for different operands). Section 3.4 contains a guide to the literature on systolic computing.

In fact, this model can be seen as a practical application of the *cellular automata* theory, which was introduced by [von Neumann, 1951] (see also [von Neumann, 1966]). A cellular automaton is a regular network (we will

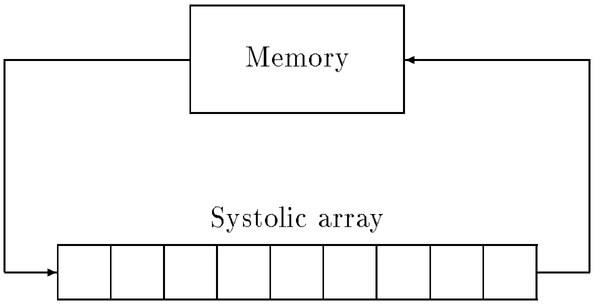

Figure 1.1: A systolic system

only consider linear arrays) of identical finite state machines (FSM). Each FSM evolution step depends on its own state as well as of the states of a finite number of neighbors (usually only the closest). More details on this model are presented in section 3.3.

One notes that the model is, in fact, quite restrictive. This is a disadvantage when designing algorithms for it, and a big disadvantage when trying to design parallelizing compilers for it (although some successful research has been going on – see e. g. [Dadda and Wah, 1993]). However, these restrictions have the advantages:

- the model has efficient practical implementations (SIMD, hardware);
- it is easy to solve problems like synchronization, deadlock elimination, routing, which are difficult on less restrictive models.

Consequently, if a good systolic algorithm **is** found, then it will be quite robust and most likely it will have a very efficient implementation.

Algebraic computations constitute the context of our approach (see also fig. 1.2). In the sequel we provide short description of the symbolic computation field and a rationale for our approach.

The need for *exact computations* is becoming more and more acute both in theoretical Mathematics and Computer Science and in industrial applications of these fields. Typical applications of symbolic computation are *Gröbner bases* [Buchberger, 1985a] (which is used, for instance, in solving systems of polynomial equations) and *quantifier elimination* [Collins, 1975] (which is used, for instance, in robot kinematics). Recent years have witnessed a continuous increase of the interest in this field, both from theorists

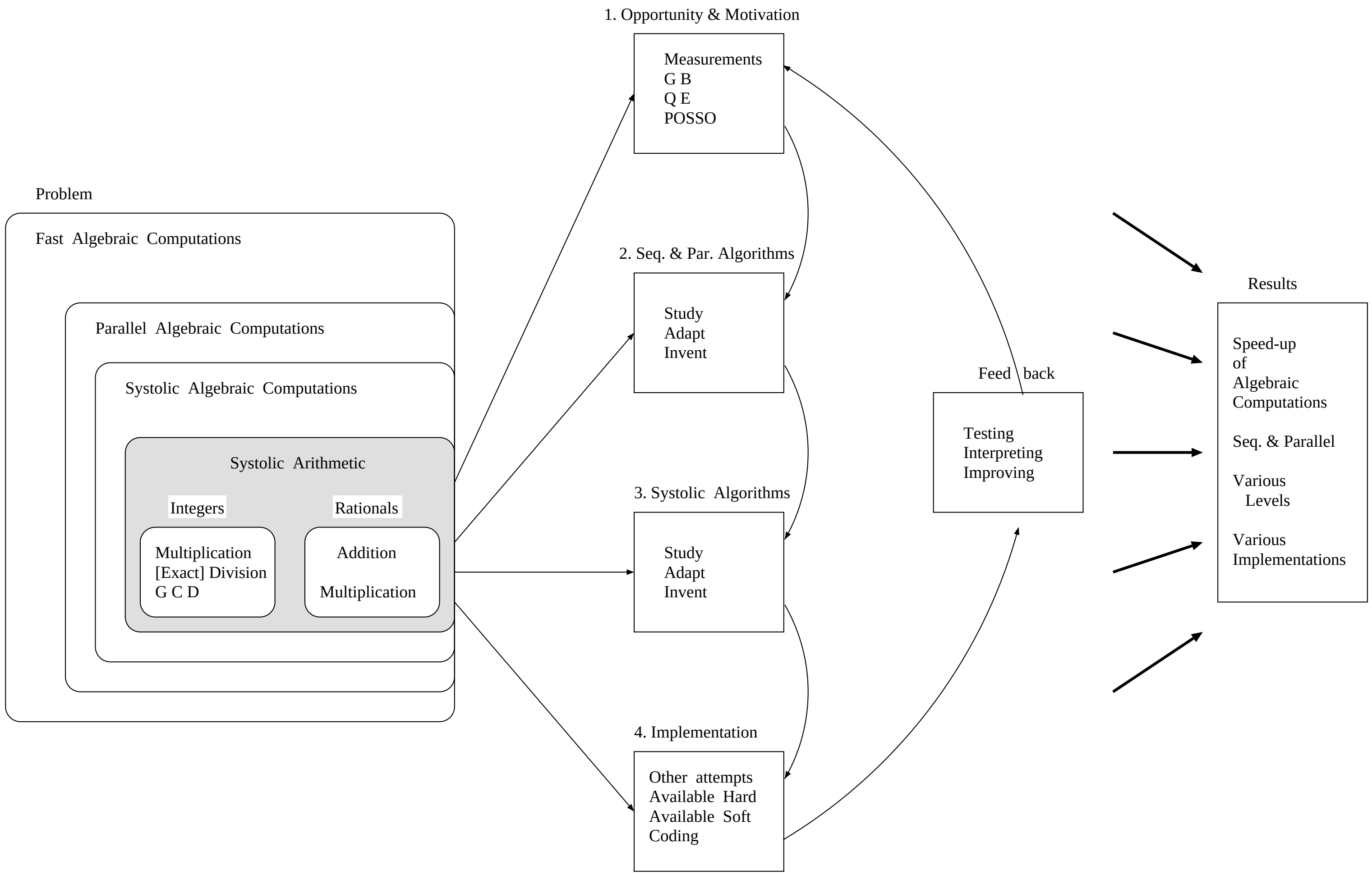

Figure 1.2: Research plan.

and practitioners (see e.g. [POSSO, 1992, Wang, 1992, Bronstein, 1993, Cohen, 1991, Cohen and van Gastel, 1992, H. F. Mattson, 1991, Cohen *et al.*, 1993]). A large collection of advanced computer algebra algorithms has already been created, which practically covers all the necessities (see e.g. [Buchberger *et al.*, 1982, Davenport *et al.*, 1988, Akritas, 1989]). Most of these algorithms are implemented in computer algebra systems available on all popular computers (Mathematica, Maple, Reduce, Axiom, Derive, etc.). However, the day-to-day use of these algorithms for actual problems occurring in scientific research and design has a major draw-back: the computing time spent in solving almost any non-trivial problem is so long, that in many cases the use of computer algebra systems is impractical. Therefore, any significant speed-up of computer algebra systems will not only result in an increasing user-commodity, but also in a significant enlargement of the area of tractable problems.

The use of coprocessors for solving basic and intensive computational tasks (as 32- or 64-bit integer arithmetic, floating-point arithmetic, graphics display, etc.) is largely spread in computing technology. Perhaps the most relevant example is floating-point arithmetic, which was once programmed/microprogrammed, then implemented in coprocessors and today it is embedded in the standard CPU of most modern computers. This evolution is due to the fact that, in recent decades, the most expensive practical computation tasks were of numerical nature.

When approaching computational problems which require arbitrary precision arithmetic, the natural question arises: Why not have specialized coprocessors for this kind of arithmetic? Certainly the frustration of the users of current computer algebra systems is at least partially due to the absence of such devices from their computers. What would be the performance of a contemporary numerical system if the floating-point operations were implemented in software?

The possibility of using special hardware for expensive algebraic computations has been already stated in [Davenport and Robert, 1985]:

> In computer algebra systems (such as REDUCE and MACSYMA), the expressions manipulated are large and complicated, for example rational functions in several variables, and hence the algorithms suffer, in general, from a relative slowness (even when they are implemented on the most powerful computers).
>
> Could one hope to increase the capacity or speed (or both) of these systems by implementing certain basic functions in hardware? We

> are not talking here about completely re-writing a computer algebra system, truly a major task, but adding certain specialized processors to increase the speed of critical algorithms. The progress of VLSI technology means that such implementations are **now** possible.

Also, in the authoritative overview report [Boyle and Caviness, 1990], page 58:

> If custom chip design and manufacture become as easy as making photographic copies, special chips for important kernel components of symbolic computation may be justified.

In fact the recent developing technology of Field Programmable Gate Arrays (FPGA), also referred to as Field Programmable Logic, already allows rapid design and realization of custom chips (see, for instance [Moore and Luk, 1993]).

It is also important to emphasize the fact that arbitrary precision arithmetic is particularly expensive. In chapter 2 we describe some experiments which show that sometimes 99% of the computing time is spent in long arithmetic. Thus by realizing a customized efficient implementation of long integer and long rational arithmetic, a significant improvement of the overall system will be achieved.

1.3 Overview

This section provides a short description of the chapters of the thesis. For the impatient reader, an abstract is provided at the beginning of each chapter. We also tried to make each chapter as much as possible "self-contained", in order to help the readers who are interested only in some particular aspects of this research.

The ordering of the chapters reflects the history of this research, which took about two years to complete. Also, most of the chapters are based on individual research papers which were produced during the research, some of them presented at conferences or published in journals (see Appendix A).

The research started with the "motivating experiments" presented in **Chapter 2**. The results show that in some expensive algebraic computations (e.g. Gröbner bases), the percentage of the time spent in long rational arithmetic tends to be very high (up to 99.7%). The total computing time increases by a factor of 40 when increasing the input coefficients length from

5 to 50 decimal digits. We noticed that this increase is *exclusively* due to long rational arithmetic.

The distribution of time-weights among various operations led us to consider *rational arithmetic* as the primary goal of our research, rather than long integer multiplication, as it initially planned.

We proceeded then to the study of sequential, parallel, and in particular systolic algorithms for long integer arithmetic, the result of which is **Chapter 3**. We found that all the algorithms we need have been parallelized systolically: addition, multiplication, division, GCD, but there are two main problems when trying to use them for our purpose:

- *Addition, multiplication,* and *GCD* only have efficient systolic algorithms which work *least-significant digits first*, because of the carry propagation; while *division* only works *most-significant digits first*. This makes it impossible to aggregate these algorithms in an efficient manner, that is, to pipeline the digits between different stages of computation.

- The [unique] Brent-Kung algorithm for systolic *GCD* computation is designed at *binary* level (processors work on bits), which makes it inefficient for multiprocessor systems (which work on words). Also, this algorithm is designed for *fixed-size operands*, for the use of cryptography applications, while we need to work with variable-size operands.

The first problem is solved in **Chapter 4**, were we develop a novel algorithm for *exact division* of integers, i. e. division when it is known in advance that the remainder is null (as it happens, for instance, in normalization of rational numbers). For this particular case of division a kind of "playing back" of multiplication is possible, starting from the least-significant digits. This algorithm also takes advantage of the fact that the remainder is null, for reducing the number of digit multiplications, hence it is faster than the usual division scheme even in the sequential case (2 times if divisor and quotient are balanced).

However, most important for our research is the fact that exact division can be now performed systolically in least-significant digits first fashion. This allows us to break with the tradition of the so called "on-line" arithmetic (which is used for systolic operations on fixed point numbers, in most-significant digits first manner, using redundant number systems) and to implement simpler algorithms which are particularly efficient for long integer arithmetic. In **Chapter 5** several systolic algorithms for exact division are presented, exhibiting various pipelining properties which make them suitable for one particular implementation or another.

The next few chapters are concerned with GCD computation, which is the most complex integral operation. **Chapter 6** introduces a novel algorithm for GCD computation, which is a generalization of the *binary* scheme of Stein, and of the *plus-minus* scheme of Brent-Kung. Both these schemes work at binary level, while the new one works at word level, thus being suitable for systolic parallelization on multiprocessor machines. Even in the sequential implementation this scheme works 2.5 times faster than the traditionally considered most efficient Lehmer-Euclid algorithm.

The two reasons for this speed-up are: *halving* the number of coarse steps and *approximative* computation by retrieving only one operand in each coarse step. In **Chapter 7** we show that by using these principles the Lehmer-Euclid can be also improved by the same factor.

Chapter 8 contains a experimental comparison of the known GCD algorithms (in the sequential version), showing that the two new algorithms are in fact the fastest. Surprisingly, the experiments also show that the *binary* GCD algorithm and its improvement for multiprecision computation perform better than their Euclidean counterparts.

With **Chapter 9** we start to be concerned about *implementation* issues. First we analyze and evaluate several computer architectures from the point of view of fine-grain parallelism. This investigation leads to the decision of experimenting with the systolic algorithms studied in previous chapters:

- at word level, on the MasPar SIMD architecture;
- at bit level, on Field Programmable Gate Arrays, by implementing the algorithms directly in hardware.

Chapter 10 describes a first attempt at implementing systolic arithmetic. Namely, we implement long-integer multiplication, and also combine it with integral univariate polynomial multiplication, in a "two-level" systolic algorithm suitable for the MasPar 2-dimensional mesh architecture. As expected, the systolic implementation exhibits linear speed-up (almost 400 in the most favorable case).

Chapter 11 presents a more elaborate approach, namely we implement the rational operations needed in inter-polynomial reduction. This contains implementation of all the integral operations (addition, multiplication, GCD, exact division). They all exhibit linear scalability, hence we believe that an implementation of Gröbner bases algorithm on top of them could perform well on this architecture.

This chapter presents a first version of systolic integer GCD computation. This novel systolic algorithm solves the problems which we encountered in the Brent-Kung algorithm:

- Based on the generalized binary scheme, this algorithm also works for high-radix numbers (operates on words), hence is suitable for implementation on multiprocessor machines.

- The data is pipelined in such a way that the algorithm will perform well on variable-size inputs: while in the old algorithm the operands are piped through a stream of $4n$ processors, in the new algorithm the operands are steady in a stream of n processors, and the computation time depends linearly on the actual length of the inputs.

The second test-bed of our algorithms is hardware implementation, which is presented in **Chapter 12**. We use programmable logic in order to realize a 4-layer systolic array for rational addition, which implements the long integer multiplications and the addition, the GCD computation, and the two exact divisions needed to normalize the result. The communication with the host is fully pipelined through one end of the array, while the communications between the cells of the array are only local, making it fully scalable by simple tiling. The estimated speed-up obtained by software simulation of the circuit is in the range of k times for k-word operands, hence we believe that a very efficient "coprocessor" for rational arithmetic can be obtained.

This chapter presents a second version of systolic integer GCD computation: this time no global broadcasting is used. The GCD algorithm works also at bit-level, like Brent-Kung algorithm, however it solves the problems of termination and sign detection, which are not solved in the old algorithm. It is also suitable for use on variable-length operands: the running time depends on the actual size of the inputs. We improve the area consumption by a factor of 12 (n processors with 8 registers each, against $4n$ processors with 24 registers each), while keeping the running time at the same level.

1.4 Acknowledgments

Bruno Buchberger is the man who initiated this research, and who patiently and skillfully guided me through it, backing me all the time with his deep vision of Mathematics and with his inexhaustible enthusiasm. Thank you, Bruno!

Besides showing an infinite patience in correcting my mistakes of all kinds, George Collins made suggestions which lead to significant improvements of the material – especially Chapter 4. This work incorporates much of his knowledge and scientific expertize.

Several other people helped me very much by reviewing various parts and making very useful suggestions: Hoon Hong, Franz Winkler, Werner Krandick, Carla Limongelli. The discussions which I had during the meetings of the Parallel Lab, Computer Algebra Seminar, and Computer Algebra Colloquium have been essential. Many thank are due to the people who participated.

In fact, I can hardly think of any member of RISC which did not help me at least once during my work here. I believe the institute is a wonderful environment for doing research, and this not only because the very good infrastructure which we have here, but primary and mainly because of the excellent *people*.

And, last but not least, I wish to warmly thank my family, Crista and Andreea, for bearing with me the burden of overwork, and for forgiving to me the time which I, many times painfully, have stolen from them.

Chapter 2

Motivating experiments

We report on a preliminary statistical analysis of algebraic computations that shows the dominating effect of integer/rational number arithmetic in certain time-consuming computations.

We observed that, with increasing length of the input coefficients, nearly all the time spent in typical large algebraic computations is actually consumed by rational number operations. For example, in Gröbner bases computation, for inputs of 5-decimal-digit coefficients, the percentage is already 86%, for 50-digit coefficients it grows to 99.7%. Also, the absolute computing time increases more than 40 times, which is exclusively due to integer/rational operations.

2.1 Introduction

The opportunity of carrying out our project relies on whether a significant speed-up of the computer algebra systems could be achieved by only speeding up the integer/rational arithmetic component of the system. Obviously this depends on the proportion of time spent within this component. Also, by investigating the growth of the total computation time when the length of the numbers involved increases, one may obtain a good judgement about the possible speed-up.

Few papers deal with this kind of experiments.

[Weeks, 1989] notes that in Gröbner bases computation "by far the largest computation time was being spent in" integer arithmetic, with the GCD computation "the most demanding of all". The experiments reported in [Neun and Melenk, 1990] find that 80% of the computation time is spent with rational arithmetic. Also, [Lazard, 1992] reports on applications of Gröbner bases in robot kinematics which lead to polynomial coefficients with more than 925 decimal digits.

Our experiments were done using `saclib` (see [Buchberger *et al.*, 1993], the C-implementation of the well known computer algebra system SAC-2 [Collins and Loos, 1982]). The hardware was a Digital DEC-station 5200 (RISC architecture, 25 MHz, 24 MIPS).

Analyzing Gröbner bases computation and related algorithms we observed that most of the computing 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 account of integer/rational operations.

The experiments used various techniques for measuring the computation time and for tracing the running algorithms:

- system clock;
- repeated call of the sub-algorithm measured;
- book-keeping of input arguments for separate evaluation;
- run-time tracing of routine-calling, time spent and input arguments.

The results were checked for correctness by comparing the output of different methods.

The measurements were done at different algorithmic levels of the computer algebra system:

- Long integer multiplication (straightforward digit by digit) takes from 0.66 milliseconds (100 decimal digits) to 55 milliseconds (1000 digits). For this range no speed-up is possible by using algorithms with better theoretic complexity (Karatsuba, FFT).
- Important integral polynomial algorithms such as *factorization* (IPFAC) and *remainder sequence* (IPSPRS) heavily rely on long integer multiplication (IPROD) and division (IQR). The combined time-weight of IPROD and IQR increases from 25% to 85% within IPFAC and from 33% to 90% within IPSPRS when increasing the input coefficient length from 3 to 150 decimal digits. Other algorithms (polynomial GCD, polynomial resultant), which are implemented using modular arithmetic, spent only less than 10% of the time in long integer arithmetic.
- Some high level algorithms, such as Gröbner bases, spent most of the computing time within integer/rational arithmetic. For instance, 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 from 0.8 to 21 seconds. Also, by tracing in detail the behavior of the algorithm for various inputs, we observed that the dramatic increase of the total computing time is exclusively due to the increase of the input coefficients length.

2.2 Timing Techniques

This section presents the techniques used for timing the computations.

The Unix system provides several functions for time tracing of programs: `prof`, `gprof`, `profil`, `monitor`. However, they are not well suited for our purpose:

- The results of these measurements are only "statistically" correct, since the timings are done by inspecting the program counter at regular time intervals (1/60 seconds).
- As in any other computer algebra system, `saclib`'s basic operations are list manipulations, which are called by all the other (e.g. integer arithmetic) algorithms. The use of `prof` will just indicate that most of the time is spent within these list operations. Even the use of `gprof`, which tries to give for each routine the total time spent including the

subroutines, is not useful, because `gprof` computes these times by using the static routine calling graph of the program. Or, in `saclib`, all the routines share the same list operations, hence this approach does not give a significant result.

- It is not possible using these tools to hide certain routines which are not of interest for us.

Of course, one could still use these tools by making significant changes in the source code. We preferred, however, to implement our own tools, which requires the same programming effort but gives us more control and flexibility.

2.2.1 System clock

For all timings we used the `saclib` routine ACLOCK, which gives the CPU time exclusive of garbage collection time. This time is obtained via the Unix call `times`, which unfortunately has an accuracy of 17 milliseconds only. This is too coarse for our needs: for instance, multiplication of integers up to 500 decimal digits takes less than 17 milliseconds. Even getting access to the native clock-tick (4 milliseconds) of the DEC-station 5200, which is used by `profil`, does not significantly increase the resolution (240 decimal digits integers are multiplied in 4 milliseconds).

Therefore, one cannot use a straightforward approach in order to measure the time taken by various sub-algorithms of a running program, hence we had to use various other timing schemes.

In one case (run-time tracing of routine calling) we used the ACLOCK time, as approximative as it is, for getting the time spent in each routine. This is because, on one hand, a more exact timing in this case would be too difficult to implement, with respect to our main research theme. On the other hand, the results obtained are accurate enough for our purpose. This tracing was done for a high level algorithm (Gröbner bases computation), which calls many times (up to 1000) the basic arithmetic routines, and we are interested mainly in the sum of all the times for each routine, rather than the individual time of each call. One can use statistical arguments to prove that in such a situation the sum of the coarse measured times is a good approximation of the real sum. We also checked the correctness of the results by repeated runs and by sample cross-checking with other timing methods.

2.2.2 Repeated call

In order to find the computation time needed for long integer multiplication of various length integers, we just repeated the computation until the total time was significantly higher then the clock resolution.

Also, in order to find the time taken by a specific integer operation, say long integer division, within a higher level algorithm, say polynomial factorization, we modified the source code of the integer division routine, by inserting a supplementary call of integer division with the same arguments. Hence, the obtained program calls executes two times each division. The difference between the running time of this program and the running time of the initial version gives the time spent within division.

2.2.3 Book-keeping

Perhaps the most accurate measurement method is to keep a trace of the arguments of each call, and then use this trace to compute separately the time spent with each call. However, this process is quite tedious and time consuming. We made use of it only for sample cross checking of the results obtained by other methods.

2.2.4 Run-time tracing

This method was used for tracing Gröbner bases computation. We developed a set of routines for tracing and timing, and we inserted calls to these routines at the beginning and at the end of each function of the algorithm which was subject to tracing. The routines display the time spent within each function (*without* traced sub-functions) and some information about the inputs to each function. The timing provided by the tracing routines is corrected so that it does not include the time spent within the tracing routines themselves. This timing is only "statistically" correct, as we mentioned in a previous subsection. However, the method allows one to spot the time consuming parts of the algorithm quickly and to trace the interesting routines, because it provides flexible ways of turning the tracing on or off, both at compile time and at run time. Also, it allows the tracing of the arguments of each function call, in order to find out which characteristics of the arguments have a higher impact on the computing time.

```
BETA-      Milliseconds
digits  Classic/Karatsuba

  2:  0.049  0.055 o
  4:  0.135  0.143 o
  6:  0.265  0.275 o
  8:  0.440  0.447 o
 10:  0.657  0.663  o
 12:  0.916  0.933  o
 14:  1.217  1.225  o
 16:  1.567  2.455   o
 18:  1.967  2.856   *+
 20:  2.389  3.333   *+
 22:  2.867  3.833    *+
 24:  3.400  4.333    *+
 26:  3.943  4.917     *+
 28:  4.555  5.500      *+
 30:  5.250  6.335      *+
 32:  5.917  9.362       *  +
 34:  6.695 10.082        *  +
 36:  7.443 10.887        *   +
 38:  8.280 11.663         *   +
 40:  9.167 12.557          *   +
 42: 10.000 13.387           *  +
 44: 10.943 14.220            *  +
 46: 12.250 15.165             *  +
 48: 13.085 16.085              *  +
 50: 14.335 17.085               *  +
 52: 15.335 18.085                *  +
 54: 16.420 18.920                 *  +
 56: 17.750 20.080                  * +
 58: 19.085 21.085                   * +
 60: 20.420 23.920                    *   +
 62: 21.585 28.500                     *      +
 64: 23.170 31.670                      *       +
 66: 24.670 33.170                        *      +
 68: 25.830 34.510                         *      +
 70: 27.670 35.840                           *     +
 72: 29.340 37.330                            *     +
 74: 30.830 38.500                              *     +
 76: 32.500 39.830                                *    +
 78: 33.830 41.000                                 *    +
 80: 35.670 42.830                                   *    +
 82: 39.000 43.670                                      *  +
 84: 39.670 45.330                                       *  +
 86: 41.000 46.680                                        *   +
 88: 43.320 48.320                                          *   +
 90: 44.660 49.680                                            *   +
 92: 47.660 51.320                                               * +
 94: 49.000 52.680                                                *  +
 96: 51.000 54.000                                                  * +
 98: 53.340 55.340                                                    * +
100: 55.320 57.340                                                     * +
```

Figure 2.1: Multiplication time for classic and Karatsuba algorithm.

2.3 Results

2.3.1 Long integer multiplication

Because long integer multiplication is one of the most costly operations, we first measured the time spent by this algorithm and the possibilities of speed-up. The present algorithm uses the straightforward digit-by-digit multiplication. Plotting the multiplication time against operand length we obtained the expected parabola (0.7 milliseconds for 100 decimal digits, 14,3 ms for 500 dd, 55.3 ms for 1000 dd). We also measured the time taken by Karatsuba algorithm [Karatsuba and Ofman, 1962], finding that it tends to become lower than the initial algorithm only from 1000 decimal digits upwards. Figure 2.1 shows the comparative timings of the two algorithms. A "BETA-digit" is a one–digit number in the radix used by `saclib` (2^{29}), and roughly corresponds to 9 decimal digits.

Also, we found out that the time needed for accessing the operands and building the result in the memory takes 1/10 (for 100 decimal digits) to 1/100 (for 1000 decimal digits) of multiplication time. This may be seen as an upper bound of possible speed-up using an external hardware device for multiplication only. The detailed results are listed in figure 2.2.

2.3.2 Algorithms on integral polynomials

Typical subalgorithms used by most computer algebra applications were also subject to experiments investigating the proportion of time spent in integer arithmetic. The results show:

- Linear complexity algorithms (as summation, difference, negation) consume an insignificant time of the computation (up to 5%).
- Multiplication (IPROD) and division (IQR) consume a significant part of the computation in some basic algorithms as IPFAC (integral polynomial factorization) and IPSPRS (integral polynomial subresultant polynomial remainder sequence). The proportion of time spent in IPROD and IQR increases dramatically when the length of the coefficients grows (see Fig. 2.3 and Fig. 2.4).
- Integer arithmetic consumes an insignificant time (up to 10%) within the algorithms which rely on modular arithmetic, as IPRES (integral polynomial resultant) and IPGCD (integral polynomial GCD). Actually, these algorithms use modular arithmetic because the integer

```
BETA- Milliseconds  Fraction
digits  access/total

   2: 0.012  0.049    1/4 *
   4: 0.022  0.135    1/6   *
   6: 0.031  0.265    1/8    *
   8: 0.040  0.440   1/11      *
  10: 0.050  0.657   1/13        *
  12: 0.058  0.916   1/16          *
  14: 0.068  1.217   1/18            *
  16: 0.078  1.567   1/20             *
  18: 0.087  1.967   1/23               *
  20: 0.096  2.389   1/25                *
  22: 0.106  2.867   1/27                  *
  24: 0.114  3.400   1/30                    *
  26: 0.125  3.943   1/32                     *
  28: 0.133  4.555   1/34                      *
  30: 0.143  5.250   1/37                        *
  32: 0.151  5.917   1/39                          *
  34: 0.161  6.695   1/42                            *
  36: 0.172  7.443   1/43                            *
  38: 0.180  8.280   1/46                              *
  40: 0.190  9.167   1/48                                *
  42: 0.200 10.000   1/50                                 *
  44: 0.207 10.943   1/53                                   *
  46: 0.219 12.250   1/56                                     *
  48: 0.227 13.085   1/58                                      *
  50: 0.235 14.335   1/61                                        *
  52: 0.244 15.335   1/63                                          *
  54: 0.252 16.420   1/65                                           *
  56: 0.262 17.750   1/68                                             *
  58: 0.275 19.085   1/69                                              *
  60: 0.265 20.420   1/77                                                   *
  62: 0.290 21.585   1/75                                                  *
  64: 0.302 23.170   1/77                                                   *
  66: 0.307 24.670   1/80                                                     *
  68: 0.320 25.830   1/81                                                      *
  70: 0.327 27.670   1/85                                                        *
  72: 0.340 29.340   1/86                                                         *
  74: 0.347 30.830   1/89                                                           *
  76: 0.357 32.500   1/91                                                            *
  78: 0.367 33.830   1/92                                                             *
  80: 0.373 35.670   1/96                                                                *
  82: 0.383 39.000  1/102                                                                     *
  84: 0.397 39.670  1/100                                                                   *
  86: 0.400 41.000  1/102                                                                     *
  88: 0.411 43.320  1/105                                                                       *
  90: 0.422 44.660  1/106                                                                       *
  92: 0.433 47.660  1/110                                                                          *
  94: 0.439 49.000  1/112                                                                           *
  96: 0.450 51.000  1/113                                                                            *
  98: 0.467 53.340  1/114                                                                             *
 100: 0.467 55.320  1/119                                                                                *
```

Figure 2.2: Fraction of time consumed for accessing the operands and the result in long integer multiplication.

arithmetic routines are so time consuming. It may be that by using a high speed unit for integer multiplication and division, the old straightforward versions of these algorithms will have faster implementations then the modular ones.

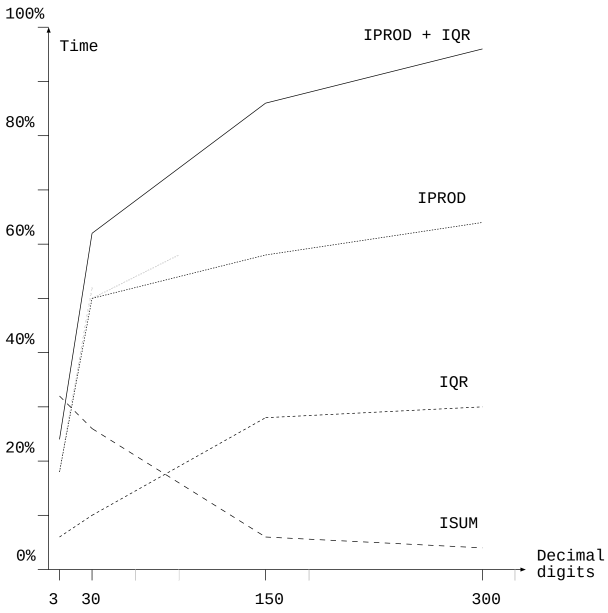

Figure 2.3: Integer arithmetic within IPFAC (plot).

Figures 2.5, 2.6 and 2.7 present the details of the measurements. The columns have the following meaning:

CoeffLength/Dec : input coefficient length in decimal digits (approximative);

CoeffLength/Bin : input coefficient length in binary digits (exact);

IPROD : time percentage spent in long integer multiplication;

IQR : time percentage spent in long integer division (quotient and remainder);

ISUM : time percentage spent in long integer summation;

Rest : time percentage spent in operations other than integer arithmetic;

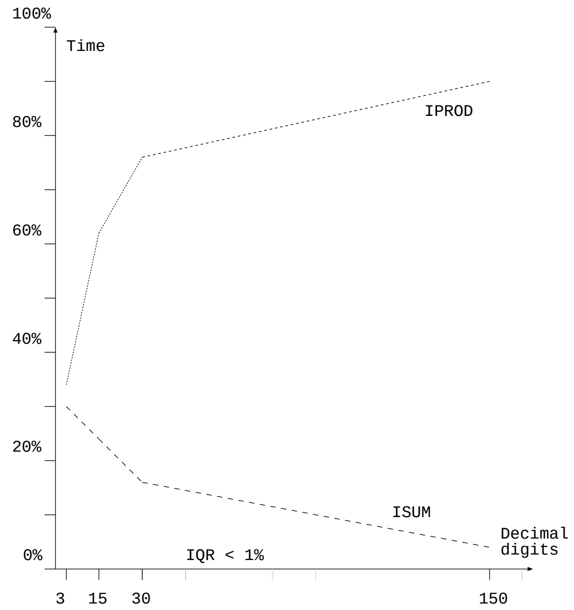

Figure 2.4: Integer arithmetic within IPSPRS (plot).

P+QR : sum of IPROD and IQR;

Var/# : number of variables of the input polynomials;

Degs : polynomial degree in each variable;

Dense : density of input polynomials;

Raw data : absolute timings;

Repeat : number of repeated computations, with different inputs of same characteristics.

The absolute timings were obtained by the "double-call" method. That is, the timings in the column **Raw data/total** are of the original `saclib` program, and the timings in the column **Raw data/IPROD**, for instance, are obtained by replacing the original IPROD routine with a double call to it, then timing the obtained program on the same inputs and subtracting the original time.

Figure 2.5 presents the results concerning the time spent by integer arithmetic within IPFAC (integral polynomial factorization). Two polynomials

CoeffLength Dec	Bin	IPROD %	IQR %	ISUM %	Rest %	P+QR %	Var #	Degs	Dense
3	10	18.5	6.3	32.6	43.0	24.9	3	2,2,2	1/2
30	100	51.5	10.4	25.9	12.1	61.9	3	2,2,2	1/2
150	500	57.3	28.6	6.9	7.3	85.8	3	2,2,2	1/7
300	1000	64.9	30.1	4.8	0.2	95.0	3	2,2,2	1/10

Raw data (milliseconds)

CoeffLength Dec	Bin	IPROD	IQR	ISUM	rest	P+QR	total	repeat
3	10	47,416	16,252	82,396	110,133	63,688	256,197	*5
30	100	272,636	55,323	137,004	64,052	327,959	529,015	*3
150	500	100,834	50,300	12,200	12,781	151,134	176,115	*2
300	1000	279,381	129,546	20,712	714	408,927	430,353	*1

Figure 2.5: Integer arithmetic within IPFAC (table).

with the indicated characteristics were randomly generated, then multiplied, and then the time taken by factorization of the product was measured.

Figure 2.6 presents the results concerning the time spent by integer arithmetic within IPSPRS (integral polynomial "subresultant" polynomial remainder sequence). Two polynomials P1, P2 with the indicated characteristics were randomly generated, then the time taken by IPSPRS(P1,P2) was measured.

Figure 2.7 presents the results concerning the time spent by integer arithmetic within IPRES (integral polynomial resultant). Two polynomials P1, P2 with the indicated characteristics were randomly generated, then the time taken by IPRES(P1,P2) was measured.

2.3.3 Gröbner bases computation

High level algorithms which are of use in typical applications are the most interesting for us. Such an algorithm is Gröbner bases computation [Buchberger, 1965, Buchberger, 1985a], which is used in solving systems of polynomial equations, quantifier elimination, and many others. We performed experiments using the implementation by B. Buchberger and W. Windsteiger [Buchberger, 1990, Windsteiger, 1990, Windsteiger, 1992] on top of `saclib`.

CoeffLength Dec	Bin	IPROD %	IQR %	ISUM %	Rest %	P+QR %	Var #	Degs	Dense
3	10	33.2	--	29.6	37.3	33.2	3	3,3,3	1
15	50	62.8	--	23.1	14.1	62.8	3	3,3,3	1
30	100	75.1	--	16.0	8.9	75.1	3	3,3,3	1
150	500	90.1	--	3.0	6.3	90.1	3	3,3,3	1

Raw data (milliseconds)

CoeffLength Dec	Bin	IPROD	IQR	ISUM	rest	P+QR	total	repeat
3	10	10,918	--	9,734	12,282	10,918	32,934	*10
15	50	82,796	--	30,482	18,524	82,796	131,802	*10
30	100	127,269	--	27,084	15,011	127,269	169,364	*5
150	500	525,178	--	17,650	36,525	525,178	579,353	*1

Figure 2.6: Integer arithmetic within IPSPRS (table).

CoeffLength Dec	Bin	IPROD %	IQR %	ISUM %	Rest %	P+QR %	Var #	Degs	Dense
3	10	0.4	1.3	0.0	98.2	1.7	3	4,4,4	1/2
30	100	0.8	4.2	1.4	93.7	4.9	3	3,3,3	1/2
150	500	1.4	1.7	5.9	91.0	3.0	3	3,3,3	1/2
300	1000	2.6	0.4	8.8	88.6	2.5	3	2,2,2	1/2

Raw data (milliseconds)

CoeffLength Dec	Bin	IPROD	IQR	ISUM	rest	P+QR	total	repeat
3	10	1,034	2,951	155	231,358	3,985	235,498	*5
30	100	2,254	11,749	3,950	265,063	14,003	283,016	*5
150	500	5,113	6,229	22,362	342,631	11,342	376,335	*1
300	1000	5,983	1,051	24,502	245,781	7,034	277,317	*3

Figure 2.7: Integer arithmetic within IPRES (table).

The Gröbner bases algorithm was applied to two dense polynomials in two variables, with the highest power products x^3y^2 and x^2y^3 (total degree lexicographic ordering, $y < x$).

When increasing the input coefficient lengths from 1 to 50 decimal digits, the computation time increases from 2 to 339 seconds, and the weight of rational arithmetic increases from 62% to 99.7%, as displayed in Fig. 2.8 and Fig. 2.10. Table 2.9 and 2.11 show the detailed timings (milliseconds).

The timings are obtained by calling ACLOCK (CPU time minus garbage collection) at the beginning and at the end of each traced routine. Since these timings are only "statistically" correct, we cross-checked them using the "double-call" method.

We also traced the different computations and compared the traces, observing that exactly the same steps are performed in each case, and the computation time can be split into:

- computation time independent of the coefficient length (operations with exponents, decisions of how the algorithm proceeds), which remains practically unchanged in all the examples (0.3 to 1.1 seconds);
- computation time dependent on the coefficient length (difference and product of coefficients), which increases very much as the length of the input coefficients grows (from 0.5 to 331.1 seconds).

This proves that the computation duration increase of 170 times is exclusively due to the rational/integer arithmetic. By speeding up rational arithmetic, let's say, 20 times, a 20 times speed-up of the Gröbner bases computation would be achieved.

Figure 2.12 shows the comparative analysis of the traces for 1 decimal digit and 5 decimal digit coefficients.

A trace contains the following information for each traced routine:

- the name of the routine at the beginning and at the end of the call;
- the "total" time spent within the routine, including the called routines;
- the "local" time spent within the routine, excluding the traced subroutines;
- the arguments of the routine;
- trace of *if* statements reached within the routine;

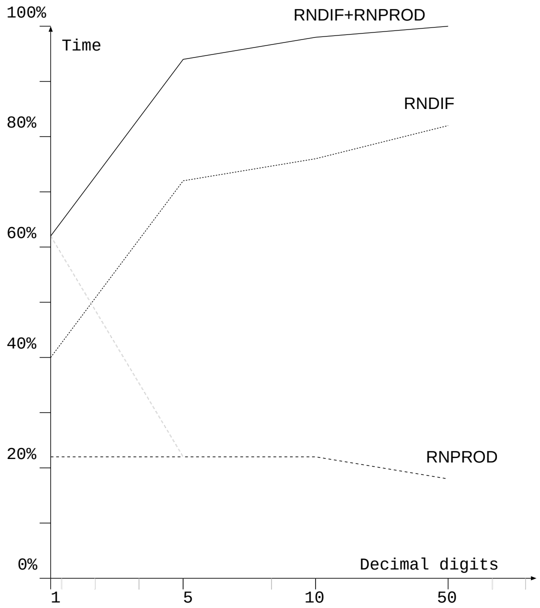

Figure 2.8: Rational arithmetic within Gröbner Bases (plot).

Length dec dig	RNPROD %	RNDIF %	Rest %	RNPROD+DIF %
1	22.1	40.3	37.6	62.4
5	21.6	72.8	5.6	94.4
10	22.5	75.3	2.2	97.8
50	17.5	82.2	0.3	99.7

Raw data (milliseconds)

Length dec dig	RNPROD	RNDIF	Rest	RNPROD+DIF	total
1	184	336	313	520	833
5	1,638	5,536	426	7,174	7,600
10	4,800	16,063	466	20,863	21,329
50	58,062	273,096	1,142	331,158	332,300

Figure 2.9: Rational arithmetic within Gröbner Bases (table).

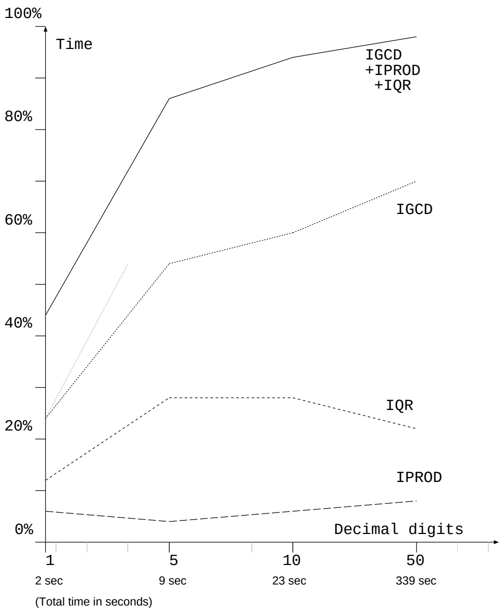

Figure 2.10: Integer arithmetic within Gröbner Bases (plot).

1 dec digs		5 dec digs		10 dec digs		50 dec digs		
total:	2040	total:	9160	total:	23150	total:	339082	
51%	1058	11%	1053	5%	1263	0%	1225	rest
3%	62	0%	84	0%	163	0%	518	INEG
1%	36	0%	59	0%	116	0%	534	ISUM
6%	134	4%	447	5%	1358	7%	23982	IPROD
12%	253	28%	2625	27%	6345	21%	74414	IQR
24%	495	53%	4890	60%	13903	70%	238407	IGCD
43%	882	86%	7962	93%	21606	99%	336803	IPROD+IQR+IGCD

Figure 2.11: Integer arithmetic within Gröbner Bases (table).

- trace of *while* statements reached within the routine, including the trace of each loop.

Choosing of the traced routines was done as follows:

- The routines whose behavior does not depend on the coefficients were not traced "inside" (called subroutines, *if* statements, *while* statements). Examples of such routines are:
  - `is-greater-mon`: comparison between two monomials with respect to the total-degree lexicographic order;
  - `product-monic-mon-dpl`: product of a power product and a polynomial ("shifting");
  - `criteria`: decides whether the S-polynomial of two polynomials should be computed.
- The routines whose behavior depends on the coefficients were also traced "inside", down to the level of rational arithmetic operations. Examples of such routines are:
  - `GB-reduce-all`: the main routine of GB algorithm;
  - `complete-normal-form-poly`: reduces a polynomial with respect to the existing base;
  - `difference-dpl`: difference of two polynomials.

The comparative analysis is done by a program which:

- Checks whether in the two traces the same routines are called in the same order, and the same *if* and *while* statements occur, with the same number of loops.
- Compares whether the arguments of each call are identical.
- Counts the number of calls of each routine and sums the total and local timings.
- Outputs the results in a table like figure 2.12.

# Calls SameArgs	# Calls DiffArgs	Time in trace1 total	Time in trace1 local	Time in trace2 total	Time in trace2 local	Routine name
0	1	1538	0	15610	0	GB_reduce_all
0	1	0	0	0	0	normalized_ps
0	7	0	0	249	0	normalized_dpl
0	5	0	0	0	0	RNINV
0	5	0	0	249	0	product_coef_dpl
0	1073	202	202	3739	3739	RNPROD
1	0	0	0	0	0	labeled_ps
0	4	34	0	100	0	all_reduced
6	0	0	0	0	0	merge
2	0	17	0	49	0	normal_form_lp
21	0	0	0	0	0	unlabeled_ps
20	0	1488	0	15210	0	complete_normal_form_poly
0	382	1488	0	15210	0	partial_normal_form_poly
7	0	0	0	0	0	split
0	5	0	0	0	0	thinned_pairs
2	0	0	0	0	0	insert_lps
465	0	0	0	0	0	is_multiple_mon
2974	0	117	117	116	116	is_greater_mon
61	0	0	0	0	0	quotient_monic_mon
0	53	285	0	3607	0	product_mon_dpl
0	968	252	50	3590	84	product_mon
968	0	0	0	16	16	product_el
0	57	1102	0	11637	0	difference_dpl
978	0	820	34	11269	50	difference_mon
0	978	786	33	11219	33	RNDIF
0	1050	50	50	16	16	RNNEG
0	978	703	703	11170	11170	RNSUM
0	28	0	0	17	0	negative_dpl
71	0	0	0	0	0	CONC
0	4	0	0	0	0	updated_pairs
14	0	17	0	51	0	head_reduced_lp
0	14	17	0	51	0	head_reduced_poly
1	0	17	0	51	0	completely_reduced_lps
10	0	0	0	0	0	criteria
0	4	50	0	151	0	S_polynomial_poly
4	0	0	0	0	0	lcm_monic_mon
8	0	34	34	0	0	product_monic_mon_dpl
72	0	0	0	0	0	negative_mon

Figure 2.12: Comparative trace analysis of Gröbner Bases for 1 vs 5 decimal digits.

2.4 Conclusions

The investigations presented in this chapter prove the usefulness of designing and implementing systolic algorithms for integer and rational arithmetic, to be used by computer algebra systems. They also give an idea about the speed-up which could be achieved.

The memory (lists) accessing time sets an inconvenient bound on the speed-up for any arithmetic operation we might consider to be performed by an additional hardware unit. For instance, in long integer multiplication, 1/10 (for 100 digits) to 1/100 (for 1000 digits) of the computation time is spent for loading the operands and storing the result. Therefore, the use of a separate coprocessor is not worthwhile for linear time operations like integer summation or negation, and it is not very efficient for other integer operations like integer multiplication or division. Rather, one might expect a very high speed-up by implementing *rational arithmetic*, because then several integer operations can be performed, without having to access the main memory for the operands of each operation, but only for the initial/final data. Since systolic algorithms are well suited to pipelining, the intermediate results of the computation inside the special unit can be pipelined between the different modules. Also, one may implement systolic algorithms for specific operations needed in rational arithmetic, rather then implementing modules for the basic integer arithmetic only. For instance, the cellular multiplier described in [Katona, 1982] is also capable of computing $a*b+c*d$ in the same number of steps as necessary for computing $a*b$.

Chapter 3

Survey

We survey the literature on integer arithmetic, that is: addition, multiplication, division, and GCD computation, with special emphasis on systolic algorithms. A separate section describes the attempts at *practical* realization of devices for systolic arithmetic.

Many results are very useful for our research, but some algorithms will need redesigning in order to fit the particular requirements of symbolic computation.

3.1 Goal of the survey

This chapter will refer to *cellular automata*, *systolic arrays* and *arithmetic algorithms* on these models, as well as to *practical implementations*. It is not meant to cover *all* aspects of the research in these fields, but rather to evaluate the present research results from the point of view of our approach. That is, we intend to identify those results which are helpful for designing and implementing systolic algorithms for long integer and long rational arithmetic, and then we consider *how helpful* these results are and what is still missing in order to reach our goal. This will raise some problems which we then try to solve in the rest of the thesis.

Let us remember that our problem is working with *integer* and *rational* operands of *variable* and *very big* size. The first of the above characteristics means that we are interested in integral algorithms which perform well when they are aggregated the way it is needed in rational arithmetic. The *variable size* characteristic of our problem imposes the use of algorithms whose running time depends on the *actual* size of the input. This is different from what is needed in other areas (for instance, in cryptography).

For algorithms targeted at *hardware* implementation, the need for *long operands* forbids the use of *2D arrays*, which would be too expensive to implement in practice. Also, one has to avoid *global broadcasting*, which could introduce significant delays.

Working with *very long* operands brings into discussion the use of *multiprocessor* machines. Systolic algorithms can be efficiently implemented on SIMD architectures (such as MasPar) or on MIMD distributed memory if the processors have very fast communication capabilities (such as transputer networks). Hence, one needs good algorithms which work well at *word* level.

3.2 Non-standard arithmetic systems

We will also set here an important characteristic of our problem: the numbers will be represented in the classical *non-redundant positional notation*. There are some interesting alternatives to this natural scheme:

- redundant signed-digit arithmetic;
- modular techniques;
- computation with continued fractions.

We shortly discuss in the sequel some advantages and disadvantages of these techniques in comparison to the classical approach.

The first efficient scheme for **redundant signed-digit** arithmetic was proposed by [Aviziensis, 1961] and had later many variants with corresponding algorithms for all the basic arithmetic operations (see collection of representative papers in [Schwartzlander, 1990a], chapter 3). The basic principle is to store an additional amount of information in each digit (a sign or a carry), and in this way to limit the effect of carry propagation to a constant number of digits (usually 2). In this way parallelization and pipelining in *most-significant digits first* manner becomes possible, giving birth to the so called *on-line arithmetic* (see overview in [Ercegovac, 1984]). The disadvantages are that the space needed for storing the operands increases, and also the algorithms become more complicated (especially for division). Note also that signed-digit arithmetic is pipelined most-significant digits first, while conversion to non-redundant representation can be done only least-significant digits first. This means that whenever a conversion is necessary, the pipelining chain must be broken, with negative impact on performance. The research presented in this thesis shows that least-significant digits first algorithms are more appropriate for dealing with *integers*, in contrast to most-significant digits first algorithms, which are more appropriate for *fixed-point* arithmetic.

Modular techniques are based on computation in homomorphic images, and can be seen as particular instances of the *parallel p-adic* scheme [Colagrossi and Limongelli, 1988, Limongelli, 1993a]: Each integer (or rational) number is represented as a collection of truncated power series (Hensel codes) of order r for n single precision primes $p_1, \ldots, p_n$ (each prime corresponds to a homomorphic image). The computations are carried out in each homomorphic image, and then the result is mapped back into the true result by use of CRA (Chinese Remainder Algorithm) and Extended Euclidean Algorithm.

If $r = 1$ and $n = 1$, then the backward mapping is trivial, but this can be applied only to a small class of algorithms – for which we know in advance that the result is single precision. In some situations it has been shown how to *lift* the modular result to the true result by repeated application of a sophisticated technique of finding $n \bmod p^{k+1}$ from $n \bmod p^k$ (or $n \bmod p^{2k}$ from $n \bmod p^k$ in *quadratic* lifting – see [Zassenhaus, 1981]). The lifting algorithm, however, is intimately related to the particular algorithm to which this technique is applied, hence a special research (usually quite nontrivial) has to be carried out for each algorithm (see e. g. [Wang, 1981, Wang *et al.*, 1982]).

If only $r = 1$, than one has the *residue number system* (RNS) arithmetic – see [Soderstrand *et al.*, 1986]. Versions for rational numbers have been studied by [Svoboda, 1962, Banerji and Kaushik, 1983, Gregory and Krishnamurthy, 1984].

If only $n = 1$, than one has the *truncated p-adic arithmetic* – see [Mahler, 1973, Yun, 1976, Gregory and Krishnamurthy, 1984]. Finally if both r and n are nontrivial, than one has the *parallel p-adic* approach, which better suited to parallelization (see [Limongelli, 1993b]).

By these techniques, the time cost of integer multiplication and rational operations becomes linear in the length of the operands (the need for rational normalization is eliminated). This may lead to a spectacular increase of efficiency in some situations.

However, comparison and integer division and are quite difficult to perform. Also, the recover step (Chinese remaindering and extended Euclidean) are computationally expensive. Therefore, the mere application of this scheme to each individual operation will be inefficient. Hence one has to *modify the target high-level algorithm* in order to make use of this technique.

Also, choosing the values of n and r at the beginning depends on the a priori bound on the length of the result. Determining this bound is a theoretically difficult task for most of the algorithms, in some cases the known bounds are too loose - which leads to (practically) useless computations, in some cases (e. g. Gröbner bases computation) they are not even known.

Finally, rational arithmetic based on **continued fractions** (see [Knuth, 1981] for this notion) seemed an impractical way of avoiding the expensive normalization step, until the work of [Gosper, 1972] who develops some practical algorithms for operating in this representation. Further work on implementation (see e. g. [Seidensticker, 1983, Kornerup and Matula, 1987]) obtained quite remarkable results, but these are practically suitable only to *finite precision* arithmetic (as an alternative to fixed-point representation). For arbitrary precision arithmetic, the behavior of these algorithms is hardly predictable, since the computing time will depend on the length of the continued fractions of the operands and of the result. Also, parallelization at the level of arithmetic algorithms is quite difficult, rather one has to parallelize the higher level algorithm for which this technique is used.

The considerations above are just meant to show that a definitive answer to the question *"Which method is the best?"* cannot be given at the present

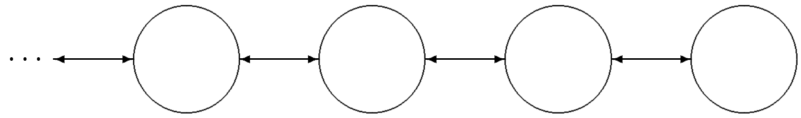

Figure 3.1: A cellular automaton.

state of the art in the field. However, we believe that a first step in studying multiprecision systolic arithmetic should try to exhaust the possibilities which are offered by the most natural approach (that is: representation of integers in the classical positional system), which has the advantage of leading to *simpler* and more natural algorithms. Study of other, more intricate schemes, comes as a natural continuation of this research, and it could also bring interesting and useful results.

3.3 Cellular automata

Cellular automata were introduced by [von Neumann, 1951] (also in [von Neumann, 1966]) as a general model of computation. The structure of such a device is remarkably simple: a regular network of identical finite state machines – "cells", yet they have maximal computing power: a linear array unbounded in one direction, with only neighbor-to-neighbor communication will be able to simulate a Turing machine (see [Herman, 1969]). There has been a wealth of literature dedicated to this model (see e. g. [Preston and Duff, 1984, Toffoli and Margolus, 1987]), maybe the most famous research is on the *game of life*, but also on formal language recognition (surveys in [Gruska, 1990, Jebelean, 1988]). Quite early the possibility of using it for arithmetic was investigated (see [Atrubin, 1965]).

From the point of view of our application, the most interesting variant of this model is the *two-way linear* device (see fig. 3.1), which is a linear array of identical cells, unbounded in one direction (this is necessary for processing arbitrary-length operands). Each cell is connected with its left and right neighbor. More formally, the transition function (which is the same for all cells) will depend on the states of the two closest neighbors, besides the old state of the cell.

There are two basic ways of computing with such a device:

- *Off-line:* The initial states of the cells represent the inputs. After a certain number of steps the result is available as the configuration (states of certain cells) of the automaton.

- *On-line:* The cells are in the same state at the beginning of the computation, but the rightmost cell receives information from the outside environment, as the state of a "ghost" neighboring cell. In this way the inputs are fed into the automaton during the computation. The result is usually extracted by the external environment by recording the sequence of states of the rightmost cell.

  This kind of device is called *iterative array* (introduced by [Hennie, 1961]) and if the computation is such that there is only a constant number of steps delay between starting the computation and starting to get the first items (digits) of the result, than the device is *on–line*.

  The most efficient variant of this kind of algorithms is the one for which the constant delay is null, and also the digits of the result are available one at each step. This is called *real-time* computation.

3.4 Systolic arrays

The term *systolic* is due to [Kung and Leiserson, 1978] (also in [Kung and Leiserson, 1980]), and it was made famous by [Kung, 1982]. These papers started a whole new area in parallel computation, which also incorporates older results on cellular automata, but it is targeted towards more practical issues (see [Fortes and Wah, 1987, Kung, 1988, Quinton and Roberts, 1989, Petkov, 1989, Bromley *et al.*, 1988, Dadda and Wah, 1993]).

The basic principle is to use a large number of [small] identical processors which are interconnected regularly and locally (usually a *mesh*) – for our application the most interesting is the linear array with neighbor-to-neighbor connections. The *locality* of the connections imposes a *pipelining* style for the communication with and within the device (see fig. 1.1): The input data is decomposed into small items (digits) which are then pipelined through the array, suffering various transformations on their way such that they finally become items of the result. This is, in fact, the reason for the name *systolic*, by analogy with the way blood circulates in an organism. The computation proceeds in discrete *steps* which are synchronously performed by all processors. At each step some operations occur on the data items and then each item moves to some neighboring processor. The terms *off-line*,

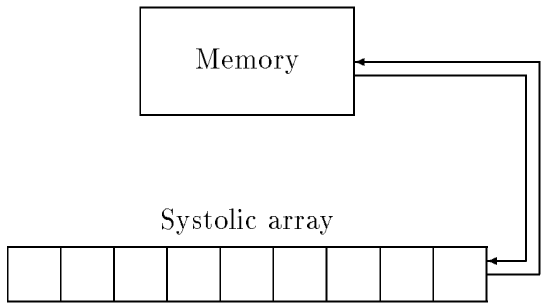

Figure 3.2: A systolic system suited for variable size problems

on-line and *real-time* have the same meaning as for cellular automata (see previous section).

Variations of the communication scheme are also possible:

- There can be several "streams" of data which are pipelined in various directions. For our approach, which applies *variable* length operands, a better scheme seems to be that of fig. 3.2, in which only one end of the array is used for communications.

- Sometimes *global broadcasting* is used in order to send values from memory (or from a certain processor) to all the processors. Broadcasting should be used with caution in order not to decrease the efficiency of the system: use it only when the target hardware system has an efficient implementation of broadcasting (e. g. SIMD architectures), and try to limit the number of broadcasting operations (e. g. to one per step).

Systolic algorithms have been developed in many areas of scientific computation, such as signal and image processing, pattern recognition, matrix and vector arithmetic, graph algorithms, etc. (see [Fortes and Wah, 1987]). We are interested mainly in *arithmetic* algorithms, which are presented in the next sections. As a general characteristic, we note that by systolic parallelization of arithmetic, the time complexity will be reduced from $O(n^2)$ (n is the size of the input in digits) to $O(n)$, with space requirements (number of processors) also $O(n)$.

3.5 Addition

It is obvious how to perform addition in a *digit serial* fashion, using only a single finite state machine (FSM). Apparently this has been first noted by [McNaughton, 1961], who also notes that multiplication *cannot* be done by a FSM. This scheme can be seen as an on-line algorithm for addition, which can be used as a computation stage between other operations, if digits are fully pipelined – for a detailed description see [Akl, 1989], section 14.2.

A truly parallel algorithm must be *off-line*, that is the digits of the two operands have to be present in the array at the beginning of the computation, each digit in one processor. After adding all pairs of digits and propagating the carries, the sum will be available in the array. The disadvantage of these scheme is that carry propagation will take (worst-case) linear time, hence no gain is obtained by parallelization. However, for applications on multiprocessor machines, this algorithm is also interesting, because the probability that the carry chain exceeds one word is very small. Hence, from the practical point of view, this algorithm will produce the result in a constant number of steps.

Carry propagation can be eliminated using redundant *signed-digit* arithmetic, but we will not investigate this line of research in this thesis (see explanation in section 3.1).

Also, we will not investigate methods which avoid carry propagation by sacrificing linear-space and regular connectivity (as e. g. carry-save, carry-look-ahead and others – see textbook [Spaniol, 1981], chapter 2, or [Schwartzlander, 1990b], part II for fixed-size addition, or [Fagin, 1992a] for multiprecision addition.

3.6 Multiplication

Long integer multiplication is perhaps the most studied of all operations, both in its sequential and parallel form.

Although the systolic algorithms are developed from the classical "school method" with $O(n^2)$ time complexity, it is worth mentioning here some of the nontrivial schemes which improve this complexity (detailed description of these algorithms can be found in [Knuth, 1981, Lipson, 1981, Aho *et al.*, 1974]).

The **Karatsuba** method introduced by [Karatsuba and Ofman, 1962] has $O(n^{\frac{\log 3}{\log 2}}) \approx O(n^{1.585})$ time complexity and it is a practical alternative to the school method even for small operands. According to the experience of the designers of GNU MP (multiprecision) package - see [Granlund, 1991], and to our own experiments, the threshold stands between 4 and 8 words (of 32 bits) for an efficient C language array based implementation. For 100 words operands the speed-up is between 2 and 2.5. Note that implementation details can affect this practical speed-up in a dramatic manner. For instance, in section 2.3.1 we notice that in SACLIB implementation, which is based on list representation of long integers, the threshold is at 100 words. However, the tree-like structure of this algorithm makes it difficult (if not impossible) to be parallelized in a systolic fashion.

k-way methods [Toom, 1963, Cook, 1966] are a generalization of Karatsuba algorithm which can be seen as a the particular case $k = 2$, and decrease time complexity even further (e. g. $O(n^{\frac{\log 5}{\log 3}}) \approx O(n^{1.465})$ for 3-way method, $O(n^{\frac{\log 7}{\log 4}}) \approx O(n^{1.404})$ for 4-way method). By suitable choosing of k in function of n one obtains a method with complexity $O(n 2^{\sqrt{2 \log n}} \log n)$ (acc. to [Knuth, 1981], section 4.3.3 A). A practical application of these methods to squaring of integers is described in [Zuras, 1993], who finds experimentally that they are practically better than FFT-methods, but worse than classical multiplication for ranges under 100 words.

Several **FFT-based** methods have been proposed [Schönhage, 1966, Schönhage and Strassen, 1971], [Lipson, 1981, section IX.2.2], decreasing the time complexity even further to $O(n \log n \log \log n)$ ($O(n \log n)$ on RAM machines. Experiments reported in [Kuechlin *et al.*, 1991, Fagin, 1992b, Zuras, 1993] show, however, that these methods are not expected to give a speed-up for the range of integers we are interested in.

Finally, the algorithm of [Schönhage, 1980] on **pointer machines** has the surprising complexity of $O(n)$, but the model itself does not seem practically realizable with the present technology.

Parallel multiplication algorithms were first studied in connection with the design of multiplication circuits for computers, already by [Booth, 1951]. A collection of milestones is [Schwartzlander, 1990b], part III and [Schwartzlander, 1990a], chapter 5. Comparative evaluation of several schemes can be found in [Habibi and Wintz, 1970], [Hwang and Briggs, 1984], section 3.2.2 and [Yasura and Yajima, 1984], section 3.1.

The lower bound for time complexity of $O(\log n)$ can be achieved by a parallel version of FFT-based multiplication method – see e. g. [Stenzel *et al.*, 1977], but the required *area* and the overhead inherent to this method makes it impractical.

On the VLSI computing model (see [Thompson, 1979, Mead and Conway, 1980], the combined lower bond for area (A) and time (T) complexity is characterized by the relation:

$$AT^2 \geq O(n^2),$$

according to [Abelson and Andreae, 1980], and more generally:

$$AT^{2\alpha} \geq O(n^{1+\alpha})$$

according to [Brent and Kung, 1981]. Circuits matching the optimal complexity are presented in [Preparata and Vuillemin, 1981, Baudet *et al.*, 1983, Luk and Vuillemin, 1983] and they are based on FFT. In connection to our problem, we may remark:

- by systolic parallelization (area and time $O(n)$) one does not obtain *optimal* cost;
- *space* requirements match, however, this optimal bound;
- any algorithm which will significantly decrease the *time* complexity under $O(n)$ will require significantly more than $O(n)$ space, which we consider impractical for our problem.

Less attention has been received by parallel methods for multiplication of *multiprecision integers* – see [Peng and Hudson, 1988, Kuechlin *et al.*, 1991, Fagin, 1992b]. However, these attempts are poorly related to our research because they use large grain parallelism and they investigate the problem for very large sizes (thousands of words). For this approach, it seems that the FFT-based methods are most efficient.

Most interesting to us is **systolic parallelization** of long integer arithmetic. An early study is [Chu, 1962], which mentions the *parallel-serial* multiplier. This scheme is a straight forward systolic parallelization of the "school method" multiplication: each step corresponds to a row of the classical method. More exactly, one operand is fed in digit-parallel fashion to the array, while the other will be fed digit-serially and broadcasted to all the processors. The operand which is present in the array is shifted one position at each step. The result can be delivered either in digit-serial or in digit-parallel manner (see section 10.3.2 for a detailed description of the algorithm).

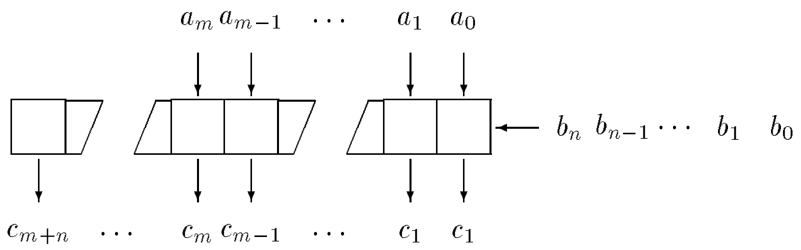

Figure 3.3: Serial-parallel multiplication.

A disadvantage of this scheme is the need of *global broadcasting*, which could lead to inefficient implementation on certain architectures. This problem is solved by a more elaborate version of parallel-serial multiplier, in which the operand which is fed in parallel will stay fixed, while the other will be pipelined *through* the array (see fig. 3.3). Note that in this variant the result will be available only after a *linear delay*.

There are many variants of this scheme, which differ in the way operands and result are pipelined (serial / parallel, most-significant digits first / least-significant digits first, leftward / rightward). It would be truly a major task to cite all the papers which treat this subject, especially because the problem is also related to *polynomial multiplication* and *convolution*. Some papers which show the developments of the idea over the years are: [Gibson and Gibbard, 1979, Hwang, 1979, Kung, 1981, Kung, 1982, Danielsson, 1984, Bucci and Di Porto, 1988, El-Desouky *et al.*, 1992]. A recent interesting approach to the construction of systolic circuits for integer arithmetic using number theory is [Vuillemin, 1993].

A distinct method is to have the two operands piped in *contraflowing* data streams [Katona, 1982, Tošić and Stoićev, 1991]. In this scheme the operands (say A, B) are intermingled with zeroes – see fig. 3.4, and it is interesting to remark that if instead of these zeroes two other operands (say C, D) are inserted, than the array performs $A * B + C * D$, which is needed in rational addition.

We would also like to make a distinct mention of *asynchronous* arrays [Goodman and McAuley, 1988, Oldfield *et al.*, 1993, Brunvand, 1993, Beister *et al.*, 1993, Planet *et al.*, 1993]. Synchronous operation seems at first glance inseparable from the systolic paradigm, but the locality of communications

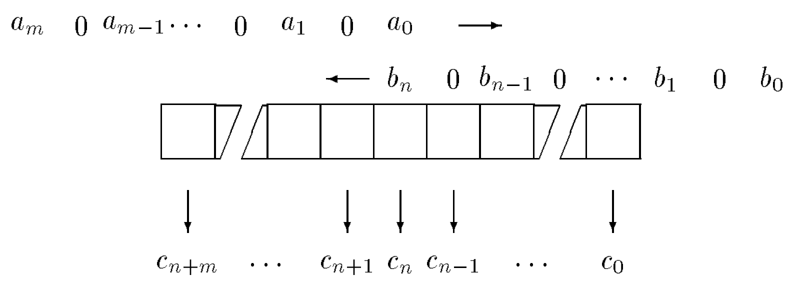

Figure 3.4: Multiplication with contraflowing operands.

makes it in fact quite possible. Two important advantages of asynchronous circuits are:

- speed increase because of self-timing: the individual modules (cells) give the response as fast as possible, since they are not constrained to the worst-case behavior;
- elimination of clock skew, which might be essential for very large circuits.

A remarkable algorithm is [Atrubin, 1965] (also described in [Knuth, 1981], section 4.3.3 E). An iterative array is used, and the algorithm is on-line and real-time. That means the operands are fed into the rightmost cell of the array, least-significant digits first, one digit of each operand per step, while the digits of the result are delivered also in the rightmost cell, one digit at each step (see fig. 3.5). The algorithm is presented for binary radix, but it can also be adapted to word-level processing in a straightforward manner. In section 5.3.5 we present an algorithm for exact division which exhibits the same pipelining properties. It is noticeable that such algorithms can be aggregated in very efficient manner in order to perform sequences of integer operations, by letting the digits flow to the next stage before the current operation is completed.

All the previously mentioned algorithms use the natural non-redundant positional representation (binary in most of the cases). We will not review here the algorithms using *signed redundant representation*, whose most efficient versions are within the scope of most-significant digits first *on-line* arithmetic. This research has also spanned a large number of papers (a

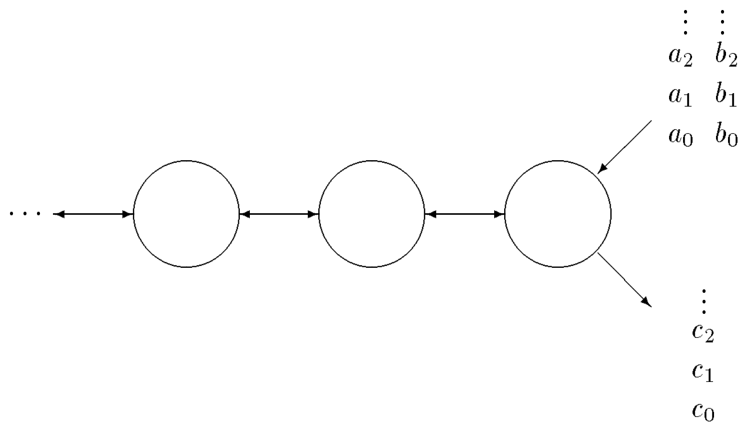

Figure 3.5: Atrubin multiplication algorithm.

good collection of milestones is [Schwartzlander, 1990a], chapter 3) but as explained in section 3.2, we prefer to pursue the least-significant digits first approach.

Looking at this wealth of literature dedicated to systolic multiplication, one may conclude that the multiplication operation is particular suitable to systolic parallelization. Therefore, one can relatively easy find a systolic algorithm which will suit the particular needs of the application, which are in our case convenient handling of input operands and efficient aggregation with the other integral operations. In sections 10.3.2, 11.2 and 12.3 one can find the particular solutions which we applied for the efficient solving of these problems.

3.7 Division

Compared to the multiplication case, the "'school method" for division of multiprecision integers is more complicated to implement in a computer program – see [Knuth, 1981], section 4.3.1. Note that in this algorithm the decisions are taken by use of the *most significant digits*. Also, each coarse step – computation of one quotient digit and updating of partial remainder – must be *completely* finished before the next step can begin, because of the carry propagation. This unpleasant property prevents the systolic parallelization of the classic division algorithm.

As for algorithms with bater theoretical complexity, the ultimate result seems to be the use of Newton's algorithm for the calculation of the inverse – see [Knuth, 1981], section 4.3.3 D. The most expensive operation within this scheme is multiplication, hence the complexity of division can be brought down to the complexity of multiplication. However, the overhead introduced by this procedure makes it impractical for the range of integers we are interested in.

As for **parallel** division algorithms, we did not find any references to the multiprecision case, but there is a lot of literature on VLSI methods – see collection in [Schwartzlander, 1990b], part IV. The complexity characterization of optimal algorithms has the same values as in the case of multiplication and leads to the same practical problems: If for word-size problems circuits matching the optimal lower bound seem practically feasible (see e. g. [Beame *et al.*, 1984]), for *large size* problems the systolic complexity (area and time $O(n)$) is apparently the only practical choice.

Systolic division is based on *redundant signed representation* and is studied in the frame of *on-line* arithmetic (see e. g. [Trivedi and Ercegovac, 1977, Ercegovac, 1984, Schwartzlander, 1990a], chapter 3). Although most of the papers consider the problem of *fixed-point* numbers, their results can be easily adapted for integer operations. The main problem for us, however, is the MSF pipelining style, since we would like to aggregate this operation with addition, multiplication, and GCD, which work *least-significant digits first*.

Fortunately, it turns out that in many algebraic computations it is not the general case division with remainder which is needed, but rather **exact division**, that is, division when the remainder is a priori known to be null. This fact has been also noted by [Weeks, 1989]: when computing Gröbner bases, only exact divisions are performed (except the Euclidean algorithm). In chapter 8 we show that GCD can be computed more efficiently without using the Euclidean algorithm.

As we demonstrate in chapter 4, exact division *can be performed* efficiently in LSF manner, and this novel algorithm can be also parallelized in systolic manner. Therefore, we investigate in the sequel the literature dedicated to exact division. It should be noted that these papers are **not** concerned with exact division. Rather, the algorithm is *hidden* among other topics. Also, they only treat the *binary* case.

Hensel odd division algorithm (acc. to [Shand and Vuillemin, 1993], section 5) is concerned with finding Q, R from M, N, such that:

$$N = -MQ + 2^p R, \quad R < 2M.$$

A systolic parallelization of this operation is presented in [Shand and Vuillemin, 1993], section 5.

[Pompeiu, 1959] (also mentioned in [Davis, 1985]) notes as a *mathematical puzzle* that when the least-significant digit of the divisor is relatively prime to 10, then one can find the quotient by using the dividend and only part of the divisor, staring from the least-significant digits.

[Purdy and Purdy, 1987] develops a systolic algorithm for division in which the remainder is found left-to-right, while the quotient is found by exact division, right-to-left, after subtracting the remainder. This is to be noted as the first (and only) attempt at **systolic parallelization** of exact division in LSF manner. We show in chapter 5 how to eliminate the global broadcasting from the Purdy scheme.

Finally, [Norton, 1992] uses an algorithm similar to Hensel for a "normalized division" between odd numbers a, b. The "normalized" q and s found by this algorithm satisfy:

$$a = qb + r2^s, \quad r < b.$$

3.8 Greatest Common Divisor

Computation of the Greatest Common Divisor (GCD) is the most complicated integral operation, and also the most time consuming in algebraic computations.

The most famous computing scheme is the *Euclidean algorithm* (see [Knuth, 1981], section 4.5.2). [Collins, 1974] proves that the maximum computing time of this algorithm is codominant with

$$n(m - k + 1),$$

where $n \geq m \geq k$ are the lengths of the operands and of the GCD. From the practical point of view, the quadratic complexity of the Euclidean algorithm has a hidden unpleasant property, which makes it distinct from the quadratic time complexity of – let us say – the multiplication algorithm. It is true that both algorithms have the same *binary* complexity, however modern computers operate with *words* of several bits. In fact, the length of the computer word is steadily increasing, which is expected to increase the performance. Multiplying two n-word integers will require n^2 word multiplications, which means that increasing the length of the computer word will reflect *quadratically* on the increase of performance. However, the number of coarse steps of the Euclidean algorithm (integer divisions) does not depend on the length of the computer word. Hence if the word length increases, then the each coarse step will be linearly faster[1], but the number of coarse steps will not change. Therefore we can expect only a linear increase of performance.

This problem was solved by [Lehmer, 1938], who found a way to simulate several division steps by using only single-precision arithmetic. Consequently, in the Lehmer-Euclid algorithm the number of coarse steps depends now on the word-length of the operands. A practical improvement of his method is due to [Collins, 1980]. In chapter 7 we show how to improve Lehmer-Euclid even further.

The running time of the Euclidean algorithm was also studied by [Brent, 1976], who finds that $0.584n$ iterations of the main loop are necessary in the average – where n is the bit-length of the inputs. The exact expression for the average number of divisions of the Euclidean algorithm is derived in [Norton, 1990].

A totally different method for computing the GCD of integers is the **binary** algorithm suggested by [Stein, 1967]. This algorithm has the same complexity, but is practically faster (see chapter 8 for a detailed description of the sequential $O(n^2)$ GCD algorithms and experimental running-time comparisons). According to [Brent, 1976], the average number of main iterations of the binary algorithm is $0.706n$ – which is more than for the Euclidean algorithm, but each iteration is simpler. This algorithm has also been improved for multiprecision computation in a Lehmer-analogous manner (see [Knuth, 1981], section 4.5.2, exercise 34) apparently by R. W. Gosper.

The *left-shift* algorithm due to [Brent, 1976] is a “left” analogue of the “right-shift” binary algorithm, which takes $0.876n$ main iterations in the average. A “least-remainder” variant of this scheme has been recently proposed by [Shallit and Sorenson, 1993].

An interesting modification of the binary algorithm is the scheme based on “normalized division” due to [Norton, 1992], which requires $0.555n$ main

[1] We think of the *time* here as being measured in number of basic computer operations, or instructions.

iterations on the average.

An unifying approach which generalizes the previous algorithms is introduced in [Sorenson, 1994], whose *left-shift k-ary* and *right-shift k-ary* algorithms apparently have sub-quadratic complexity $O(n^2/\log(n))$. However, his algorithms use some precomputed tables which make them impractical for large k's. This problem was solved (for the right-shift version) by [Weber, 1993], which designs an algorithm very similar to our *generalized binary* scheme presented in chapter 6. These last algorithms improve significantly the performance of the Lehmer-Euclid method for the range of integers we are interested in, and are also suitable for parallelization on multiprocessor machines.

The **asymptotically fastest** method is due to [Schönhage, 1971] and has time complexity $O(n \log^2 n \log\log n)$. However, this method is most probably impractical for words under 100 words. [Moenck, 1973a] generalizes Schönhage's algorithm to arbitrary Euclidean domains and shows that GCDs can be computed in time

$$O(n \log^{a+1} n),$$

using a multiplication algorithm with running time

$$O(n \log^{a} n).$$

We are not aware of any practical implementations of these algorithms.

Efficient **parallelization** of the Euclidean algorithm is difficult, because of the carry propagation. More suitable to parallelization seems to be the *binary* algorithm, because here the carries propagate "in the other direction". However, the binary algorithm requires a comparison at each step, which is also difficult to parallelize. [Purdy, 1983] tries to remove this inconvenience, but his algorithm is linear only in the average case (worst case quadratic).

The first successful parallelization, which is also **systolic** is due to [Brent and Kung, 1985], who use a modified version of the binary algorithm called the *plus-minus* algorithm. They achieve time complexity $O(n)$ with $O(n)$ processors. We would also like to note that their algorithm is *purely* systolic, i. e. no global broadcasting is used. However, there are two major inconveniences of this algorithm with respect to our problem:

- The operands are piped *through* the array and hence the worst case computing time is needed for operands of *any* length. This was in fact no problem in the context for which this algorithm was designed, which is cryptography.

- The algorithm is based on *binary*-radix operations, hence it is inappropriate for an efficient implementation on *multiprocessor* machines.

In chapters 6, and 12 we propose solutions to these problems.

A slight improvement of the Brent-Kung systolic algorithm is presented in [Yun and Zhang, 1986], but their scheme in turn uses global broadcasting, which makes it less suitable for computation with very large numbers.

A **sublinear parallel** algorithm is introduced in [Kannan *et al.*, 1987], which has time-complexity $O(n \log\log n / \log n)$. [Chor and Goldreich, 1990] improve this to $O(n/\log n)$. The parallel versions of k-ary algorithms of [Sorenson, 1994] are simpler, yet they achieve the same performance. However, the superlinear number of processors required by these algorithms makes them rather impractical.

A comparative evaluation of several **VLSI implementations** of GCD algorithms is presented in [Guyot, 1991]. Serial / parallel variants of left-shift algorithm (most-significant bits first) using *redundant signed-digit arithmetic* and serial / parallel variants of least-significant bits first schemes of Brent-Kung and Yun-Zhang schemes are investigated. All implementations use *global broadcasting.*

3.9 Systolic Devices

First let us note that there is a long history of research concerning parallel devices for *symbolic computation.* Some overviews are [Ponder, 1988, Kusche, 1991]. One of the earliest attempts was in fact developed at RISC (see [Buchberger, 1985b]).

However, most of these attempts were targeted at non-numeric algorithms – like list processing (especially for LISP systems), unification and inference (PROLOG systems) and the like.

Only recently hardware for long integer arithmetic was built:

- [Katona and Legendi, 1981] was a pioneering attempt to speed word-length addition and multiplication by using a hardware simulation of cellular automata.

- [Beardsworth, 1981] obtains linear speed-up on the DAP for univariate polynomial arithmetic, which is simpler because of the absence of carry propagation. The computation of GCD is not considered.

- [Guyot *et al.*, 1987] build JANUS for long integer multiplication and division, in LSF manner. More recently the same group reports on a signed-digit device OCAPI [Guyot and Kusumaputri, 1991] for 2000 bits on-line integer arithmetic, and also for various GCD computing experiments [Guyot, 1991]. Apparently they did not try an integrating approach to rational arithmetic. However, it is notable that the main purpose of that project is to build hardware for the use of algebraic computations.

- PERLE_0 and PERLE_1 are *active memories* [Bertin *et al.*, 1989], [Bertin *et al.*, 1992], [Shand *et al.*, 1991], that is memories which can also perform computations. By using this novel paradigm (which comes very close to our approach), the researchers in Paris DEC Laboratory implemented on arrays of FPGAs several designs solving intensively computational problems with a performance surpassing supercomputers (e. g. for cryptography). However, while PERLE family are designed as general-purpose devices, our goal is a hardware unit especially designed for multiprecision arithmetic, thus taking full advantage of our algorithmic knowledge about this problems.

- The **PAC project** [Roch, 1989], [Roch, 1990]), also intends to realize a "parallel computer algebra co-processor" (excerpt from the title of [Roch, 1989]). However, in the PAC project, the term "co-processor" designates a shared-memory parallel computer with several transputer units, together with specialized software for solving certain basic algebraic computations, including, but not limited to, arbitrary precision arithmetic. In our project, the term "coprocessor" designates a hardware unit specialized for multi-precision arithmetic. We believe that by using a specially designed hardware we can obtain a much better price/performance ratio than can be obtained by using a general-purpose computer.

- **Cascade** from [Carter, 1989] handles variable precision integers in MSF manner using custom-designed 64-bit arithmetic units. It does not address rational arithmetic and GCD computation is performed in the classic (quadratic time) fashion.

- The **MGAP** from [Irwin and Owens, 1992] was used for linear-time multiplication of long integers. A more recent version [Nagendra *et al.*, 1993] is being currently implemented on Xilinx FPGAs and used for long fixed-point arithmetic.

3.10 Conclusions

After a long standing research concerning *cellular automata* and *systolic arrays*, most of the theoretical and practical aspects of these models are well understood. In particular, there is a wealth of algorithms available for *arithmetic*, and also some practical implementations.

However, most of the research on arithmetic algorithms was concentrated on *fixed-point* operations and *bit-level* arithmetic for *fixed-size* operands. Although many results of this research can be immediately used for our application (e. g. addition and multiplication algorithms), some other algorithms need redesigning.

This is due to our need for working with integer and *rational* operands of *variable* and *very big* size. The first of the above characteristics means that we are interested in integral algorithms which perform well when they are aggregated the way needed in rational arithmetic. A problem here is the *division* algorithm: While addition and multiplication are [most efficiently] performed in the least-significant digits first manner, division can be only performed in most-significant digits first fashion.

The *variable size* characteristic of our problem forbids us to use algorithms as Brent-Kung GCD, because this scheme will need the worst-case number of steps for **any** input. The GCD algorithm will need redesigning in order to be time-dependent on the *actual* size of the input.

For algorithms targeted at *hardware* implementation, working with *very long operands* forbids use of circuits having more than $O(n)$ area, which would be too expensive to implement in practice.

Working with *very long* operands also brings into discussion, the use of *multiprocessor* machines. However, here one needs good algorithms which work well at *word* level, namely for *division* and *GCD*. The known multiprecision algorithms for these two operations are not parallelizable in systolic fashion, because of carry propagation.

Looking at the **practical attempts** to implement systolic arithmetic, we note that most of the approaches are concerned with *fixed-size, fixed-point* operations. However, recently some attempts at *long integer* arithmetic have been made, which cover all the basic operations, and most of these attempts performed quite efficiently. It is notable that an attempt to realize an integrated system for *long rational arithmetic*, either in hardware or on a multiprocessor system, is apparently not yet reported in the literature. This might be partially due to the algorithmic problems which we have identified above.

Chapter 4

Exact division

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 introduce 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 is confirmed by practical experiments.

Most importantly, however, the new algorithm is better suited for systolic parallelization in a "least-significant digits 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.

4.1 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, 1981], section 4.3.1), and then just ignoring the remainder.

In this paper we introduce a novel 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 mentioned by [Pompeiu, 1959], (also in [Davis, 1985]), 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 by [Purdy and Purdy, 1987], 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, 1965], and this is very useful in constructing a systolic system for multiprecision rational arithmetic.

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

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

input operands, then the number of digit multiplications is

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

while

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

(see Collins, 1974). 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, 1991] and SACLIB Computer Algebra system [Buchberger *et al.*, 1993] show a speed-up of almost 2 for $m = 2n$, and 3 for $m = 3n/2$ (see Fig.4.3).

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).

4.2 The new algorithm

Mentioning it as a mathematical curiosity[2], [Pompeiu, 1959] notes that in certain situations one can find the least-significant digit of the quotient by using only the least-significant digits of the operands. Let A, B, C be multi-precision 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.4.1. Note that line [15] is a loop containing `Length`(A) single precision multiplications.

[2]Which, however, was never turned into a systematic algorithm.

```
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 4.1: EDIV: the algorithm for exact division.

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

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

The same algorithm can be applied to exact division of polynomials. If A, B, C are *univariate polynomials* and a, b, c are the coefficients of the least-degree monomials then a simplified version of the above algorithm will do the job (in this case there is no problem with the primality of β or with the modular inversion). I some applications, this right-to-left method might be more useful than the classical left-to-right one for certain reasons (parallelization, pipelining). It is interesting to remark that, in the case of exact division of polynomials, either left-to-right or right-to-left, some computations can be saved if one computes only the needed coefficients of the new C, as in the scheme above. The speed-up is quite high when the degrees of A and C are close (see time complexity analysis below). The same is true for exact division of *multivariate polynomials* (see also [Kolář and Sasaki, 1992]), where a similar algorithm can be applied recursively. In this case a, b, c are also *polynomials*, hence the inverse in line [7] fig. 4.1 cannot be computed. Therefore, the modular divisions in line [12] become *exact divisions* of polynomials, which can be solved by a recursive application of the same algorithm. This idea was exploited in [Kolář and Sasaki, 1992] in conjunction with power-series arithmetic.

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[3] a power of 2. In order to adapt the algorithm to this case, we introduce 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'] $\quad A \leftarrow A/2$
[5'] $\quad C \leftarrow C/2$

Let us call the classical quotient–remainder algorithm IQR. (A detailed description of this algorithm is given by [Knuth, 1981], section 4.3.1 and the time-complexity analysis is given by [Collins, 1974].) 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.)

- *Fewer digit multiplications are required.*

  In Fig.4.2 we present a pictorial "estimation" of the speed-up (shaded areas correspond to the number of digit multiplications). A detailed analysis follows below.

[3]Some interactive systems use a power of 10 as radix. This eases the conversion 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 split into two digits. If the radix is not a power of 2, this splitting requires a division.

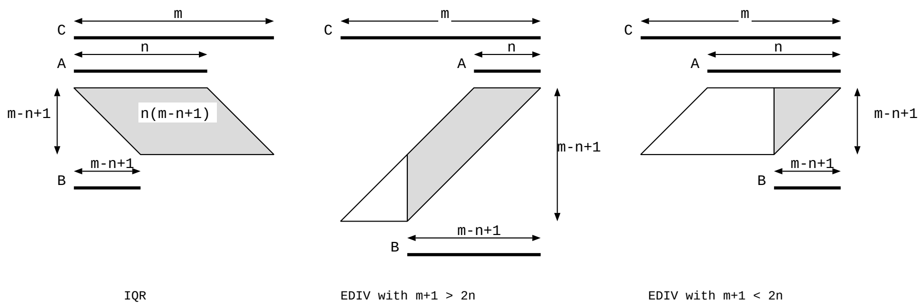

Figure 4.2: Evaluation of computing times

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

In EDIV, **if** $m+1 \geq 2n$, then $\min(n, K-k)$ 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

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

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.4.3 we present benchmarks of the new algorithm using the GNU multiprecision arithmetic library [Granlund, 1991] ($\beta = 2^{32}$) and also the

SACLIB Computer Algebra system [Buchberger *et al.*, 1993] ($\beta = 2^{29}$), with GNU C optimizing compiler on a the Digital DECstation 5200 (MIPS RISC architecture, 25 MHz, 24 MIPS). For large lengths of operands, the measured speed-up agrees 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 the 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 to decreasing the speed-up for small lengths. However, for longer integers the effect of this additional operation disappears.

4.3 GCD computation

Like the quotient-remainder algorithm, EDIV can also be used to build a GCD algorithm, which will work "least-significant digits first". This algorithm can be seen as a generalization of the GCD by "normalized division" algorithm of [Norton, 1992], which works at binary level.

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

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

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

$$0 \leq -C_{j+1} < A. \tag{4.1}$$

Furthermore, one obtains inductively

$$C_1 * \beta = C_0 - b_0 * A,$$

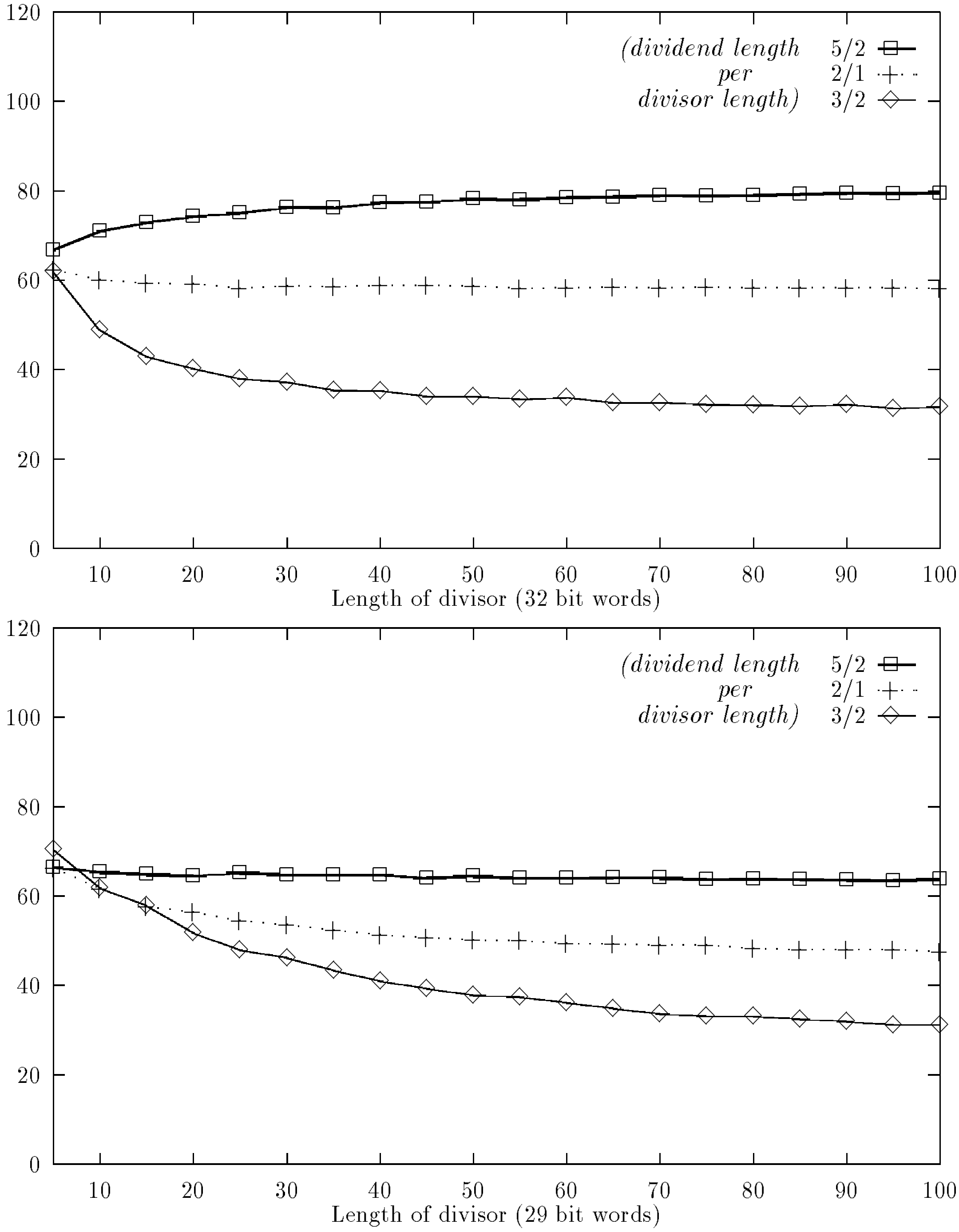

Figure 4.3: EDIV/IQR % comparative timings, under GNU (top) and SACLIB (bottom).

```
[ 0]  G ← EDGCD(C, A) [G = GCD(C_0, A), C ≥ A, β prime]
[ 1]  begin
[ 2]    G ← 1
[ 3]    [Skip common trailing zeroes]
[ 4]    while β|A and β|C
[ 5]      A ← A/β
[ 6]      C ← C/β
[ 7]      G ← G * β
[ 8]    [Main loop]
[ 9]    while A ≠ 0
[10]      [Skip trailing zeroes of A]
[11]      while β|A
[12]        A ← A/β
[13]      a' ← (LowestDigit(A))^{-1}_{modβ}
[14]      while C > 0
[15]        C ← (C − A * (a' * LowestDigit(C))_{modβ})/β
[16]      (C, A) ← (A, −C)
[17]    return G ← G * C
[18]  end
```

Figure 4.4: EDGCD: Exact division based GCD algorithm.

$$C_2 * \beta^2 = C_1 * \beta - b_1 * A * \beta = C_0 - (b_1 * \beta + b_0) * A,$$

$$\ldots$$

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

and so

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

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

$$GCD(C_0, A) = GCD(A, -C_{j+1}). \tag{4.2}$$

The relations (4.1) and (4.2) allow us to construct the algorithm EDGCD (Fig.4.4) 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 ← A/2
[ 6']        C ← C/2
[ 7']        G ← G * 2
```

and lines [11], [12] into

```
[11']      while 2|A
[12']        A ← 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 $(A, B) \leftarrow (B, A_{\mathrm{mod}\ B})$, see [Knuth, 1981], Section 4.5.3, Exercise 30), which is known to be optimal for GCD algorithms based on division (cf. [Kronecker, 1901]).

However, for practical applications this new GCD algorithm is not better than Lehmer–Euclid algorithm ([Lehmer, 1938], also [Knuth, 1981], section 4.5.2). We describe EDGCD just in order to demonstrate that the exact division algorithm can be used to devise a "least-significant digits first" GCD algorithm.

Fig.4.5 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 (32–bit words), 1000 random pairs of integers were tested. The experiments were done using the GNU multiprecision library [Granlund, 1991] and GNU optimizing compiler on Digital DECstation 5200 (MIPS RISC architecture, 25 MHz, 24 MIPS).

4.4 Division modulo 2^n

Let us consider the algorithm MODIV shown in Fig.4.6.

According to line [10]

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

and by induction

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

Length	Timings in milliseconds			Number of steps		
of operands	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 4.5: Benchmarks of EDGCD and modified Euclidean GCD.

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

Figure 4.6: MODIV: the algorithm for modular division.

where

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

Therefore

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

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. 4.7). The speed-up was measured in comparison with the extended Euclidean algorithm improved by [Lehmer, 1938] and [Collins, 1980].

For $\beta = 2$, one obtains an algorithm for division modulo 2^n which is simple enough to be suitable for hardware implementation (see Fig. 4.8). 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, 1992] notes that a very similar algorithm can be used for right to left computation of the period of the result of a fixed–point division.

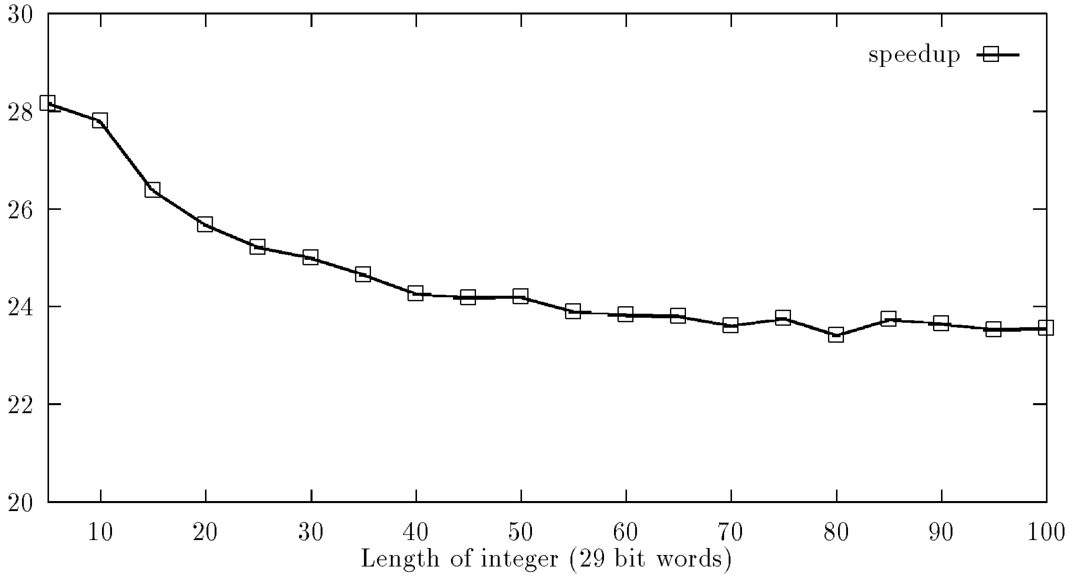

Figure 4.7: Speed up of modular inverse using SACLIB.

$B \leftarrow MODIV2(C, A)$ $[B = (C/A)_{\mathrm{mod}2^n},\ A \text{ odd}]$

```
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 4.8: MODIV2: binary algorithm for division modulo 2^n.

4.4.1 Recursive modular inversion

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

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

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

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

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).

Chapter 5

Systolic exact division

Several systolic implementations of the *exact division* 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.

5.1 Introduction

Systolic parallelization of long rational arithmetic in the most-significant digits first (MSF) pipelined manner was considered by [Trivedi and Ercegovac, 1977]. In particular, MSF systolic division was discussed by several authors [Gex, 1971, Kamal *et al.*, 1975, Preparata and Vuillemin, 1990], see also section 3.7. However, the MSF methods require redundant representations, hence additional transformation operations have to be performed. Only [Purdy and Purdy, 1987] presents a 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.

In the previous chapter we have presented a LSF algorithm for exact division based on the idea 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, 1965], GCD – [Brent and Kung, 1983]), 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.

5.2 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:

$$C = A * B.$$

5.2.1 Sequential algorithm

The basic idea of the algorithm introduced in chapter 4 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 $(C - b_0 * A)/x$.

The sequential algorithm is presented in Fig.5.1. In order to simplify the presentation, we extend A by

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

in the case $m < n - m$.

$B \leftarrow \texttt{SeqPoly}(C, A)$ [Sequential polynomial exact division]
[$n = \texttt{Degree}(C)$]
[$m = \texttt{Degree}(A)$]
[$p = n - m = \texttt{Degree}(C)$]
[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 5.1: Sequential algorithm for polynomial exact division.

5.2.2 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.5.2 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.5.3 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:

- 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.

5.2.3 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

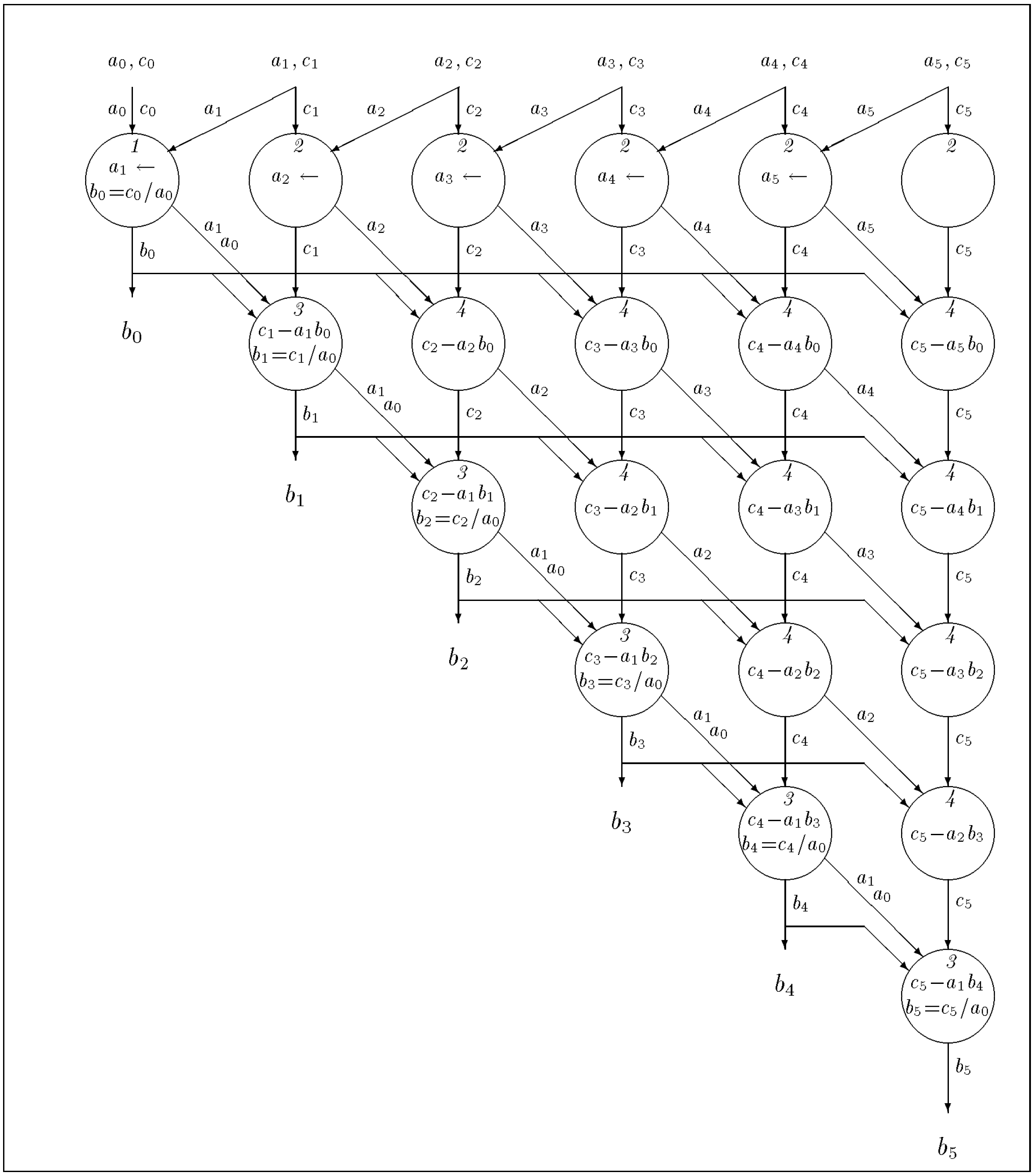

Figure 5.2: Data flow in version 0 of systolic polynomial exact division.

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

$$
\begin{array}{lllll}
\texttt{Local:} & a, a', c, s & & & \\
\texttt{Global:} & b_G & & & \\
\texttt{Input:} & a_{[k]} = a_k, & 0 \le k \le \min(n,p) & [n = \texttt{Degree}(C)] & \\
 & a_{[k]} = 0, & \min(n,p) < k \le p & [m = \texttt{Degree}(A)] & \\
 & c_{[k]} = c_k, & 0 \le k \le p & [p = n - m = \texttt{Degree}(B)] & \\
 & s_{[0]} = 1 & & & \\
 & s_{[k]} = 2, & 1 \le k \le p & & \\
\texttt{Evolution:} & \texttt{case } s = 1: & a' \leftarrow a & [a_0] & \\
 & & a \leftarrow a_{[+]} & [a_1] & \\
 & & b_G \leftarrow c/a' & [b_0] & \\
 & & s \leftarrow 0 & & \\
 & \texttt{case } s = 2: & a \leftarrow a_{[+]} & & \\
 & & \texttt{if } s_{[-]} = 2 \texttt{ then } s \leftarrow 4 & & \\
 & & \qquad\qquad \texttt{else } s \leftarrow 3 & & \\
 & \texttt{case } s = 3: & a \leftarrow a_{[-]} & [a_1] & \\
 & & a' \leftarrow a'_{[-]} & [a_0] & \\
 & & b_G \leftarrow (c - a * b_G)/a' & [b_1, b_2, \ldots] & \\
 & & s \leftarrow 0 & & \\
 & \texttt{case } s = 4: & a \leftarrow a_{[-]} & & \\
 & & c \leftarrow c - a * b_G & & \\
 & & s \leftarrow s_{[-]} & & \\
 & \texttt{case } s = 0: & \text{[no action]} & & \\
\texttt{Output:} & b_k = b_{[k]}, & 0 \le k \le p & &
\end{array}
$$

Figure 5.3: Systolic algorithm for polynomial exact division: version 0.

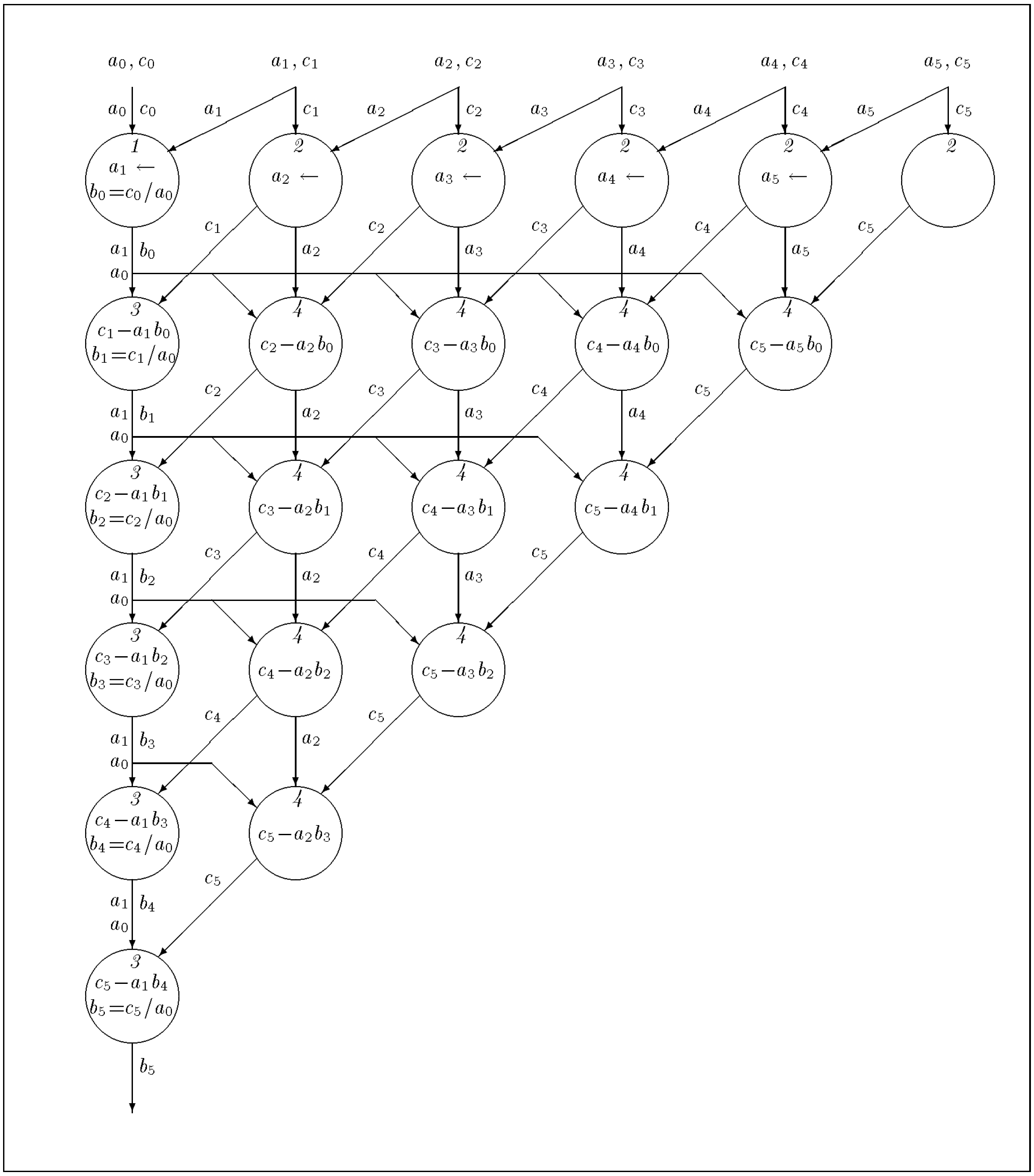

Figure 5.4: Data flow in version 1 of systolic polynomial exact division.

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

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

`Global:` b_G

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

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

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

$s_{[0]} = 1$

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

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

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

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

$s \leftarrow 3$

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

$s \leftarrow 4$

`case` $s = 3$: $b^G \leftarrow (c_{[+]} - a * b_G)/a'$

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

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

Figure 5.5: Version 1 of the algorithm for polynomial exact division.

pipelined leftward. At each step, a new coefficient of B is obtained in the leftmost cell. Fig.5.4 shows the flow of computation for $p = 5$.

The formal description is presented in Fig.5.5.

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.

5.2.4 Systolic algorithm: Version 2

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

Fig.5.6 depicts the flow of computation for $p = 5$.

The description of the algorithm is presented in Fig.5.7.

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.

5.2.5 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 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, 1965]. 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.5.8.

Before giving the formal description (Fig.5.9), let us emphasize the main ideas of this algorithm.

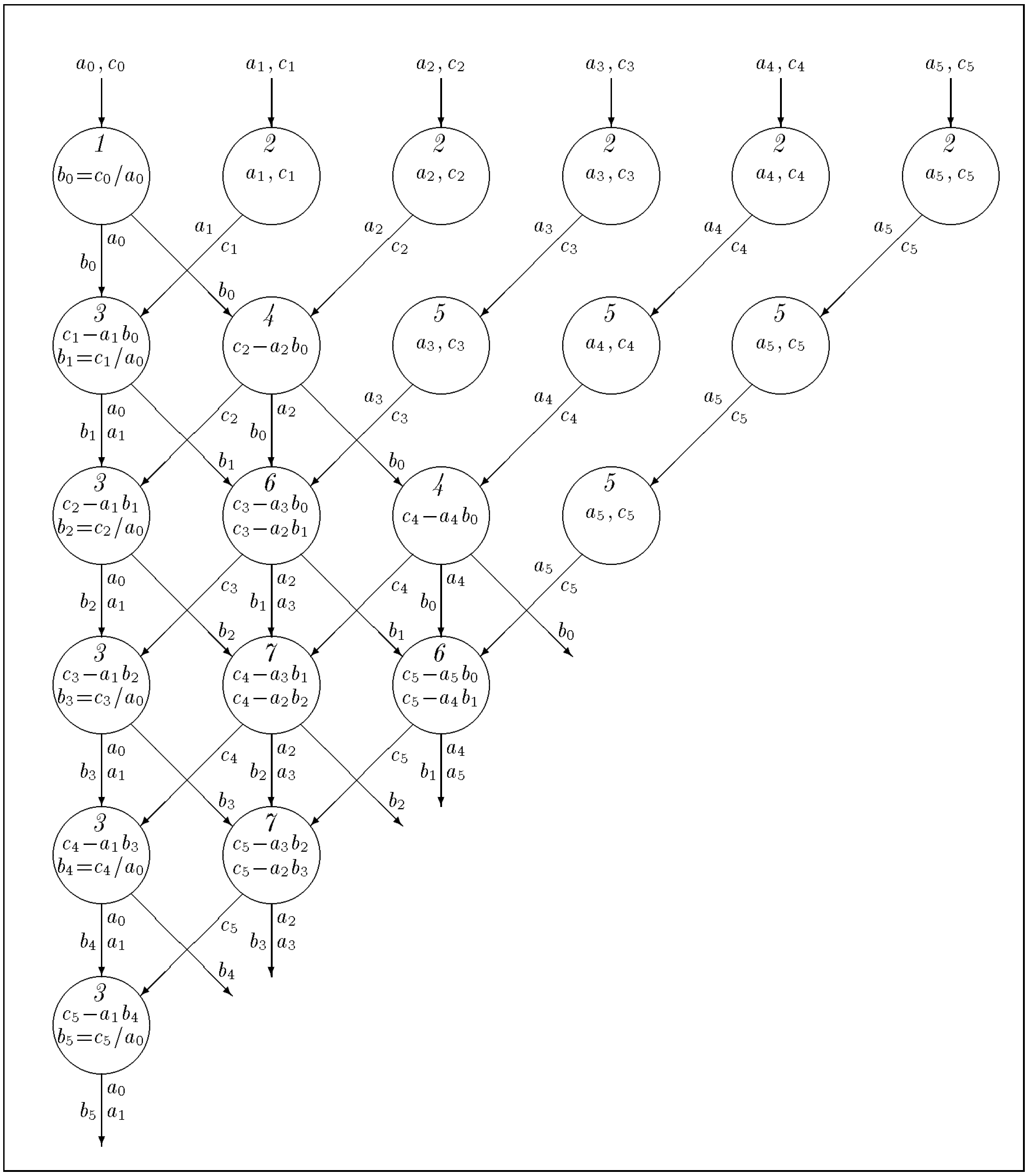

Figure 5.6: Data flow in version 2 of systolic polynomial exact division.

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

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

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

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

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

$s_{[0]} = 1$

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

`Evolution:` `case` $s = 1$: $a' \leftarrow a_{[+]} \quad [a_1]$

$b \leftarrow c/a \quad [b_0]$

$s \leftarrow 3$

`case` $s = 2$: `if` $s_{[-]} = 2$ `then` $s \leftarrow 5$

`else` $s \leftarrow 3$

`case` $s = 3$: $b \leftarrow (c_{[+]} - a' * b)/a$

[s remains 3 from now on]

`case` $s = 4$: $a \leftarrow a_{[+]} \quad$ [store a_{2k}]

$b \leftarrow b_{[-]}$

$c \leftarrow c_{[+]} - a * b$

$s \leftarrow 6$

`case` $s = 5$: $a \leftarrow a_{[+]}$

$c \leftarrow c_{[+]}$

$s \leftarrow s_{[-]}$

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

$c \leftarrow c_{[+]} - a' * b - a * b_{[-]}$

$b \leftarrow b_{[-]}$

$s \leftarrow 7$

`case` $s = 7$: $c \leftarrow c_{[+]} - a' * b - a * b_{[-]}$

$b \leftarrow b_{[-]}$

[s remains 7 from now on]

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

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

Figure 5.7: Version 2 of the algorithm for polynomial exact division.

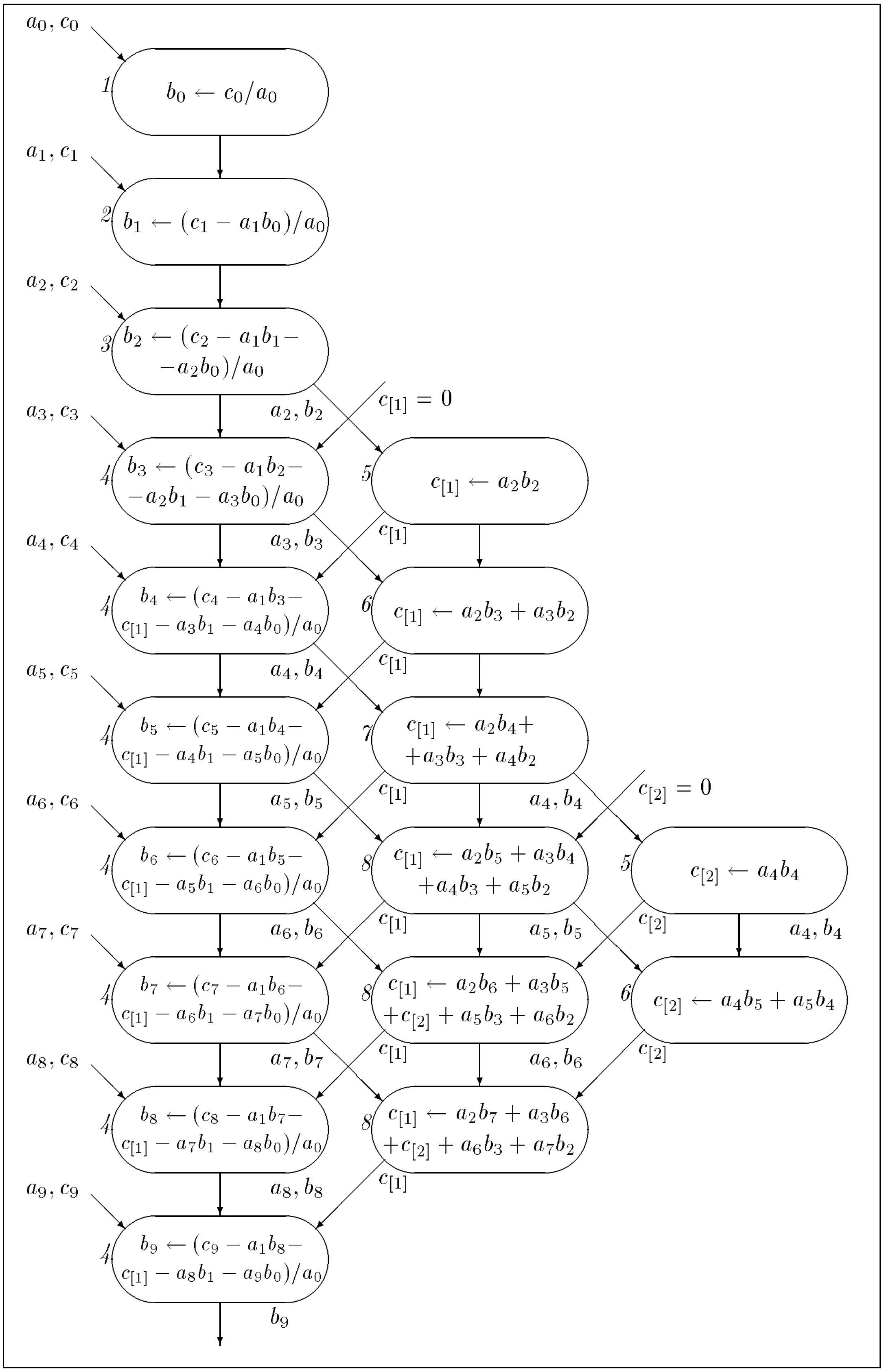

Figure 5.8: Data flow in version 3 of systolic polynomial exact division.

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.
- 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.

5.3 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

$$a_0 \neq 0.$$

$B \leftarrow$ `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 5.9: Version 3 of the algorithm for polynomial exact division.

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:

$$C = A * B.$$

5.3.1 Sequential algorithm

The basic idea of the algorithm introduced in chapter 4 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:

$$b_0 = (c_0 * (a_0)^{-1}_{\mathrm{mod}\beta})_{\mathrm{mod}\beta}. \tag{5.1}$$

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 (5.1) 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.5.10. 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.5.1), 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

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

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:

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

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

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

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 `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.5.10.

5.3.2 Systolic algorithm: Version 0

In contrast with 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.5.11.

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

Figure 5.10: Sequential algorithm for long integer exact division.

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

Figure 5.11: 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.5.12). This is similar to `SysPoly.0` (Fig.5.3), except for carry handling.

A number of $p+1$ processors are used, processor P_k containing a_k, c_k and y_k $(0 \leq k \leq p)$. 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 neighbor 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.5.13 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.

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

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

`Global:` b_G

`Input:`

$a_{[k]} = a_k,$	$0 \le k \le \min(n, p)$	$[n =$ `Length`$(C) - 1]$
$a_{[k]} = 0,$	$\min(n, p) < k \le p$	$[m =$ `Length`$(A) - 1]$
$c_{[k]} = c_k,$	$0 \le k \le p$	$[p = n - m =$ `Length`$(B) - 1]$
$y_{[k]} = 0,$	$0 \le k \le p$	
$s_{[0]} = 1$		
$s_{[k]} = 2,$	$1 \le k \le p$	

`Evolution:` `case` $s = 1$:
- $a' \leftarrow a$ $\quad [a_0]$
- $\overline{a} \leftarrow a^{-1}_{\mathrm{mod}\beta}$ $\quad [(a_0)^{-1}_{\mathrm{mod}\beta}]$
- $a \leftarrow a_{[+]}$ $\quad [a_1]$
- $b_G \leftarrow (c * \overline{a})_{\mathrm{mod}\beta}$ $\quad [b_0]$
- $s \leftarrow 0$

`case` $s = 2$:
- $a \leftarrow a_{[+]}$
- `if` $s_{[-]} = 2$ `then` $s \leftarrow 4$
  `else` $s \leftarrow 3$

`case` $s = 3$:
- $a \leftarrow a_{[-]}$ $\quad [a_1]$
- $a' \leftarrow a'_{[-]}$ $\quad [a_0]$
- $\overline{a} \leftarrow \overline{a}_{[-]}$ $\quad [(a_0)^{-1}_{\mathrm{mod}\beta}]$
- $(y, c) \leftarrow c - a * b_G -$ `High`$(a' * b_G) + y_{[-]}$
- $b_G \leftarrow (c * \overline{a})_{\mathrm{mod}\beta}$
- $s \leftarrow 0$

`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 5.12: Systolic algorithm for exact division of long integers: version 0.

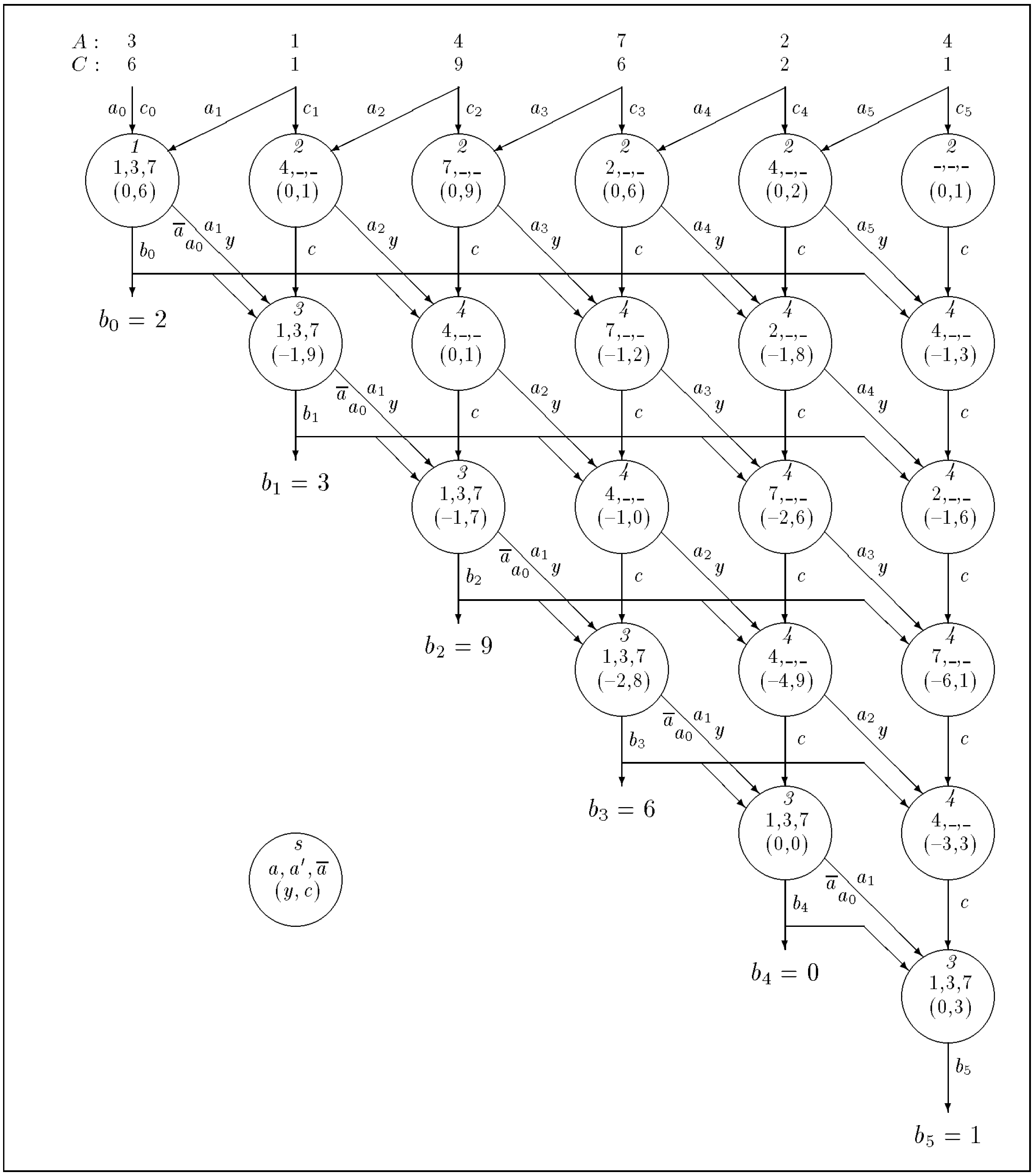

Figure 5.13: Data flow in version 0 of systolic exact division of long integers.

- During the computation, A moves rightward through the array, while C is kept steady. The carry vector Y also moves rightward through the array.

5.3.3 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 and Purdy, 1987].

Fig.5.14 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.5.15.

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.

5.3.4 Systolic algorithm: Version 2

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

Fig.5.16 depicts the flow of computation for the same example ($\beta = 10$, $C = 45,704,126,916$, $A = 427,413$, $B = 106,932$, $n = 10, m = 5, p = 5$).

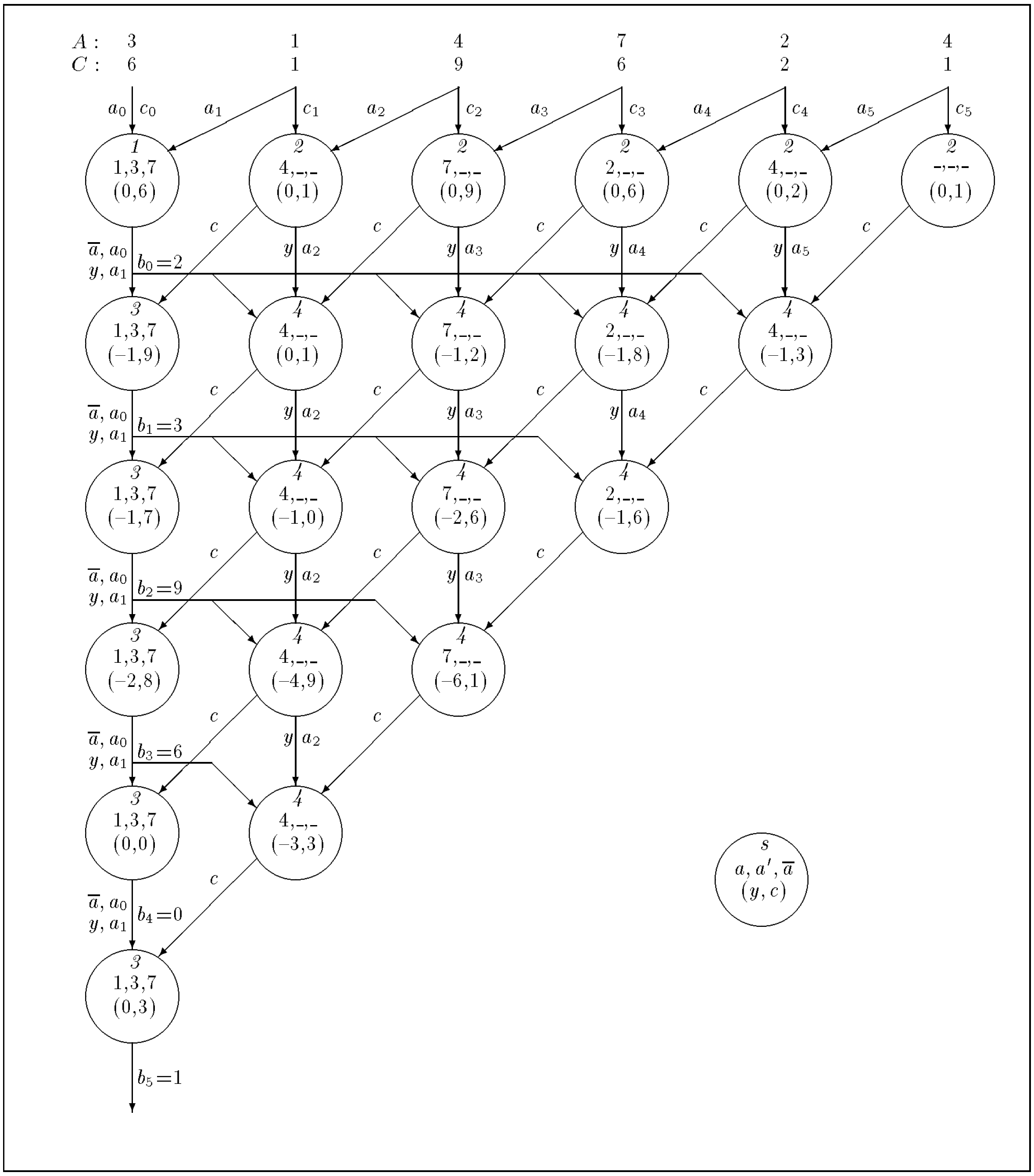

Figure 5.14: Data flow in version 1 of systolic exact division of long integers.

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

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

`Global:` b_G

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

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

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

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

$s_{[0]} = 1$

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

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

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

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

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

$s \leftarrow 3$

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

$s \leftarrow 4$

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

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

`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 5.15: Version 1 of the algorithm for exact division of integers.

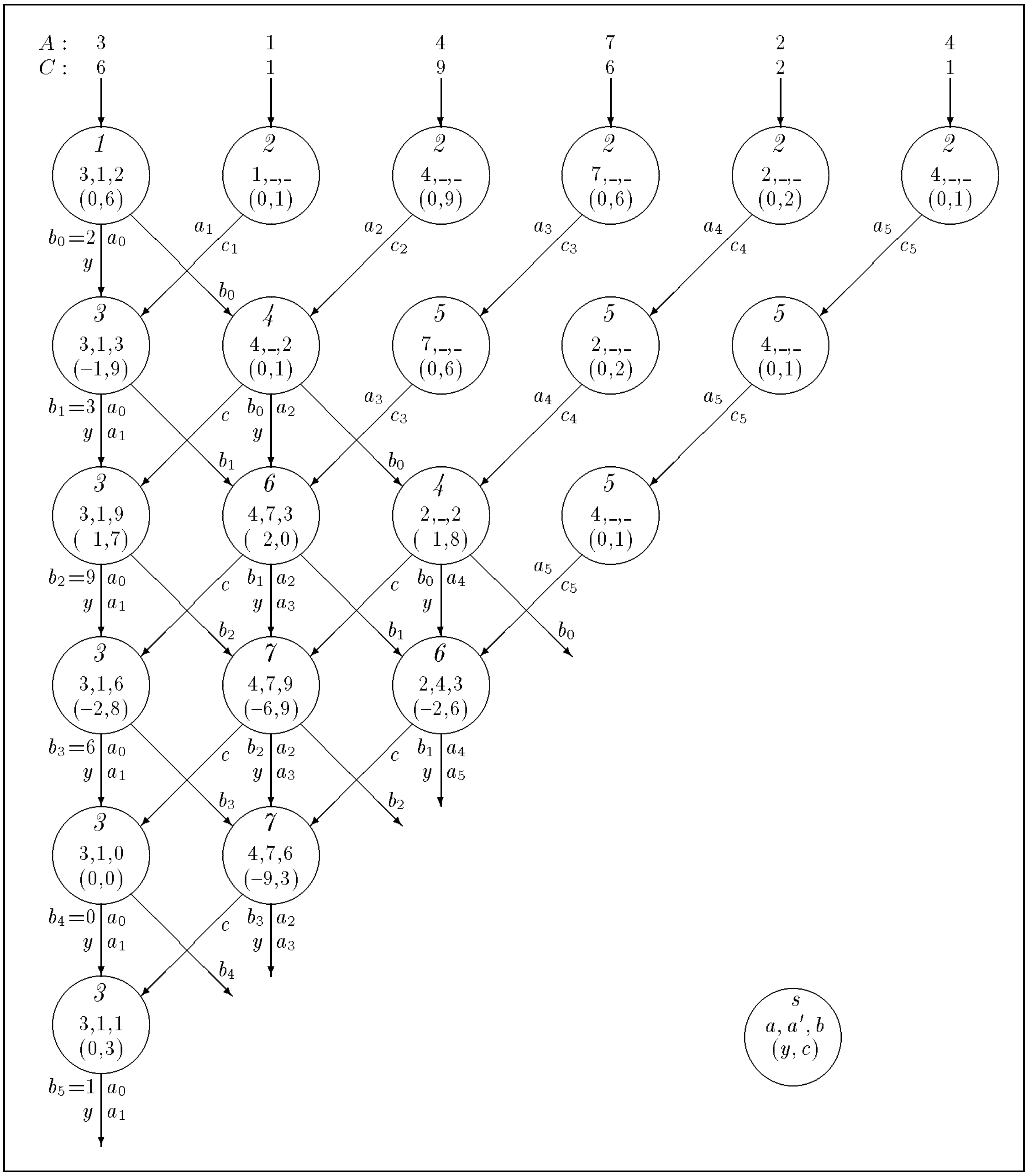

Figure 5.16: Data flow in version 2 of systolic exact division of integers.

The description of the algorithm is presented in Fig.5.17. 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.

5.3.5 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.5.18, Fig.5.19) is a modified version of the Atrubin multiplication scheme [Atrubin, 1965]. 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.

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

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

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

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

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

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

$s_{[0]} = 1$

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

`Evolution:`[states 1 and 2 are for the first step]

`case` $s = 1$: [processor 0]

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

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

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

$s \leftarrow 3$

`case` $s = 2$: [other processors are idle]

`if` $s_{[-]} = 2$

`then` $s \leftarrow 5$

`else` $s \leftarrow 4$

[subsequent steps]

`case` $s = 3$: [processor 0]

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

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

[s remains 3 from now on]

`case` $s = 4$: [first subtraction]

$a \leftarrow a_{[+]}$ [store a_{2k}]

$b \leftarrow b_{[-]}$

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

$s \leftarrow 6$

`case` $s = 5$: [just pipe a and c]

$a \leftarrow a_{[+]}$

$c \leftarrow c_{[+]}$

$s \leftarrow s_{[-]}$

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

$(y, c) \leftarrow c_{[+]} - a * b - a' * b_{[-]} + y$

$b \leftarrow b_{[-]}$

$s \leftarrow 7$

`case` $s = 7$: $(y, c) \leftarrow c_{[+]} - a * b - a' * b_{[-]} + y$

$b \leftarrow b_{[-]}$

[s remains 7 from now on]

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

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

Figure 5.17: Version 2 of the algorithm for exact division of integers.

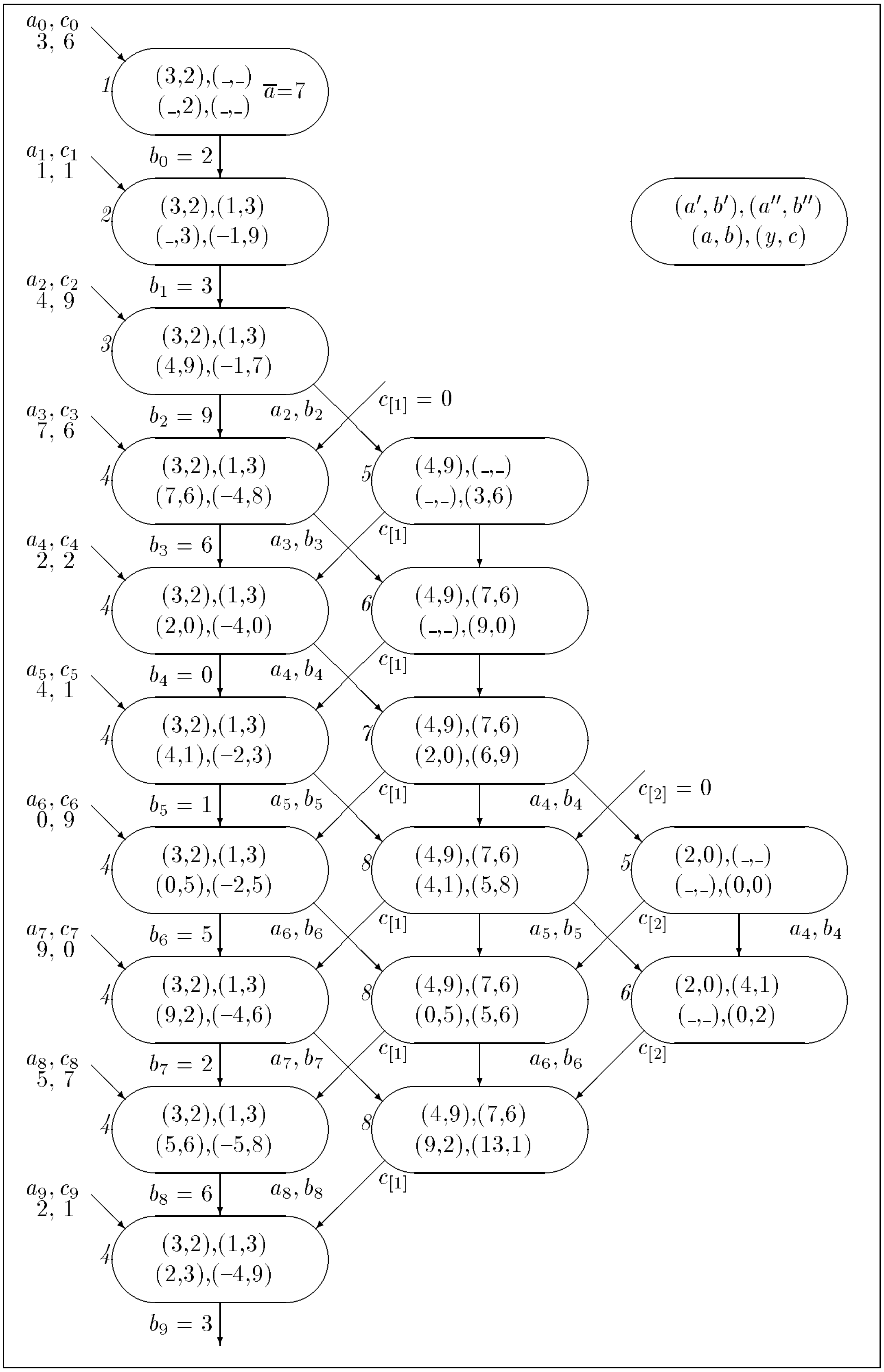

Figure 5.18: Data flow in version 3 of systolic exact division of long integers.

Fig.5.18 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.5.19. 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.

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.

$B \leftarrow$ `SysInt.3`(C, A) [Systolic exact division of integers, version 3]

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

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

`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]
$\overline{a} \leftarrow (a')^{-1}_{\bmod \beta}$ [$(a_0)^{-1}_{\bmod \beta}$]
$b \leftarrow (c_{[-]}\overline{a})_{\bmod \beta}; \quad b' \leftarrow b$ [b_0]
$s \leftarrow 2$

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

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

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

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

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

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

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

`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]

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

Figure 5.19: Version 3 of the algorithm for exact division of long integers.

5.4 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, pipelining 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.

- **Pipelining** 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.

Chapter 6

A generalization of the binary GCD algorithm

A generalization of the binary algorithm for operation at “word level” by using a new concept of “modular conjugates” computes the GCD of multi-precision integers two times faster than the Lehmer–Euclid method. Most importantly, however, the new algorithm is suitable for systolic parallelization, in “least-significant digits first” pipelined manner and for aggregation with other systolic algorithms for the arithmetic of multiprecision rational numbers.

6.1 Introduction

In this chapter we introduce new GCD algorithm which is suitable for systolic parallelization in "least-significant digits first" (LSF) pipelined manner, this time at "word" level, which makes it appropriate for implementation on multiprocessor machines. This new algorithm is a generalization of the binary GCD algorithm of [Stein, 1967] and of the Plus-Minus algorithm of [Brent and Kung, 1983].

The **binary** GCD algorithm [Stein, 1967] consists in making the two operands odd by binary–shifting, and then, since their *least–significant bits* are equal, one obtains by subtraction a number whose *least–significant bit* is null, which by another binary–shift becomes at least one-bit shorter than the original numbers. One iterates this process until the GCD is obtained. If by subtraction a negative number is obtained, then the next step will be in fact an addition, hence, in general, the result will not be shortened anymore. Therefore, at each step the two operands have to be compared in order to identify the right minuend. This makes the binary algorithm less suitable for adaptation to multiprecision computations, because such an adaptation means simulating several steps by using only the information contained in the least–significant words of the operands, and recovering the full–length operands only when simulation is not accurate anymore. Also, this makes the binary algorithm less suitable for parallelization (in particular, for systolic parallelization), because such parallelization is efficient if one "master processor" needs only the information contained in the least–significant words for making the computational decisions at each step. These decisions are then sent to other parallel "slave processor" which operate on the other digits of the operands.

In the **plus–minus** GCD algorithm, [Brent and Kung, 1983] remove this disadvantage of the binary algorithm, by noticing that, if the *least–significant double–bits* of the two (odd) operands are equal, then one can perform subtraction, otherwise addition, in order to make the *least–significant double–bit* of the result null. Hence, the shifted result is shortened no matter which are the signs of the operands. This algorithm is easier to adapt to multi-digit computation and it is also suitable for systolic parallelization (see [Brent and Kung, 1983]). However, the binary level at which this algorithm operates makes it suitable for hardware implementation, rather than for implementation on multiprocessor machines.

We show in this chapter how to **generalize** the ideas above to arbitrary bit–length: starting from the *least–significant double–words* of the (odd) operands, one can always find two cofactors ("modular conjugates") which are at most one word long, such that the linear combination of the operands by these cofactors is a number whose *least–significant double–word* is null. By binary shifting one obtains a number which is one word (for instance 32 bits) shorter then the initial operands. This algorithm is more efficient for multi-digit GCD computation than the straight forward adaptation of the plus–minus algorithm (experimentally 2.35 times speed–up), and, in fact, it seems to be faster than any other multiprecision algorithm, according to the experiments in chapter 8.

Recently other researchers found similar results. [Sorenson, 1994] proposes a "right-shift k-ary" algorithm which works in the same way, but he does not find a way to compute the modular conjugates, leaving them to table-lookup. This idea can be applied only to small k's, for 64 bits as in our implementation the size of the tables would make the scheme impractical. [Weber, 1993] improves Sorenson algorithm by giving a method to compute the modular conjugates – which is quite similar to ours, only that the *division modulo 2* is performed via an iterative algorithm of [Norton, 1989] (in fact similar to the algorithm presented in section 4.4), and not by recursion as in our implementation (see section 4.4.1). The experiments of Weber confirm our experimental results.

6.2 Modular Conjugates

Let us remember the basic Euclidean method. Starting with two positive integers $a_0 \geq a_1$, one computes the remainder sequence $(a_i)_{1 \leq i \leq n+1}$ defined by the relations:

$$a_{i+2} = a_i \bmod a_{i+1}, \quad a_{n+1} = 0. \tag{6.1}$$

and then one has:

$$GCD(a_0, a_1) = a_n.$$

A detailed presentation of the Euclidean algorithm can be found in [Knuth, 1981].

The *extended* Euclidean algorithm (also in [Knuth, 1981]) consists in computing additionally the *quotient sequence* $(q_i)_{1 \leq i \leq n}$ and the *cosequences* of *cofactors* $(u_i, v_i)_{0 \leq i \leq n+1}$ defined by:

$$q_{i+1} = \lfloor a_i / a_{i+1} \rfloor, \tag{6.2}$$

$$u_0 = 1, v_0 = 0, u_1 = 0, v_1 = 1, \tag{6.3}$$

$$(u_{i+2}, v_{i+2}) = (u_i, v_i) - q_{i+1} * (u_{i+1}, v_{i+1}). \tag{6.4}$$

Then one has:

$$a_{i+2} = a_i - q_{i+1} * a_{i+1} \tag{6.5}$$

$$u_i * a_0 + v_i * a_1 = a_i. \tag{6.6}$$

It is useful to note that the signs of the cofactors alternate. Indeed, by (6.3):

$$u_0 \geq 0, \quad v_0 \leq 0,$$

$$u_1 \leq 0, \quad v_1 \geq 0.$$

If one assumes that for any $i < k$:

$$u_i \geq 0, \quad v_i \leq 0, \quad \text{if } i \text{ even},$$

$$u_i \leq 0, \quad v_i \geq 0, \quad \text{if } i \text{ odd},$$

then using (6.4) one can deduce the above relations for $i = k \geq 2$. Hence, one has:

$$|u_i| = (-1)^i * u_i, \quad |v_i| = (-1)^{i+1} * v_i$$

and (6.4) can be also written as:

$$(|u_{i+2}|, |v_{i+2}|) = (|u_i|, |v_i|) + q_{i+1} * (|u_{i+1}|, |v_{i+1}|). \tag{6.7}$$

In order to investigate the size of the cofactors, we use the *continuant polynomials* (see also [Knuth, 1981]) defined by

$$\begin{cases} P_0() = 1, \\ P_1(x_1) = x_1, \\ P_{i+2}(x_1, \ldots, x_{i+2}) = \\ \qquad P_i(x_1, \ldots, x_i) + x_{i+2} * P_{i+1}(x_1, \ldots, x_{i+1}). \end{cases} \tag{6.8}$$

which are known to enjoy the symmetry

$$P_i(x_1, \ldots, x_i) = P_i(x_i, \ldots, x_1). \tag{6.9}$$

By comparing the recurrence relations (6.7) and (6.8) one notes

$$\begin{aligned} |u_i| &= P_{i-2}(q_2, \ldots, q_{i-1}), \\ |v_i| &= P_{i-1}(q_1, \ldots, q_{i-1}). \end{aligned} \tag{6.10}$$

Also, by transforming (6.5) into

$$a_i = a_{i+2} + q_{i+1} * a_{i+1},$$

and using $a_i > a_{i+1}$, one can prove

$$\begin{aligned} a_0 &\geq a_i * P_i(q_i, \ldots, q_1), \\ a_1 &\geq a_i * P_{i-1}(q_i, \ldots, q_2). \end{aligned} \tag{6.11}$$

Hence by (6.10) and (6.11) one has

$$\begin{aligned} |v_{i+1}| &\leq a_0/a_i, \\ |u_{i+1}| &\leq a_1/a_i. \end{aligned} \tag{6.12}$$

Let us denote by m the bit–length of the computer word (in our implementation $m = 32$), and let us consider two odd double–words a, b ($a, b < 2^{2m}$). We show in the sequel how to find the *modular conjugates* of a, b, that is, the integers x, y with the properties:

$$0 < x, |y| < 2^m,$$

$$(x * a + y * b) \bmod 2^{2m} = 0 \tag{6.13}$$

Since a, b are odd, one can find $b^{-1} \bmod 2^{2m}$ and

$$c = (a * b^{-1}) \bmod 2^{2m},$$

which is also odd. Then (6.13) becomes

$$(x * c + y) \bmod 2^{2m} = 0.$$

If $c < 2^m$, then $x = 1$ and $y = -c$ satisfy (6.13) and they also have the desired lengths. If $c \geq 2^m$, then let us apply the extended Euclidean algorithm to $a_0 = 2^{2m}$ and $a_1 = c$. Since c is odd, the remainder sequence (a_i) is strictly decreasing to $1 = GCD(2^{2m}, c)$, hence for some k:

$$0 < a_k < 2^m \leq a_{k-1}.$$

By (6.12):

$$|v_k| \le a_0/a_{k-1} \le 2^{2m}/2^m = 2^m.$$

Also, by (6.6):

$$2^{2m} * u_k + c * v_k = a_k,$$

hence:

$$(c * v_k - a_k) \bmod 2^{2m} = 0.$$

In fact, $|v_k|$ cannot equal 2^m, because in this case the above relation implies:

$$a_k \bmod 2^m = 0,$$

which is impossible for

$$0 < a_k < 2^m.$$

Hence, by taking $x = |v_k|$ and $y = (-1)^k a_k$, one has the desired modular conjugates.

The process of finding x, y from c is similar to the "integer to rational conversion" scheme mentioned in [Wang, 1981]. However, that scheme attempts to find $a, b < 2^{m-1}$, which is not always possible, while for our algorithm we prove that it is always successful.

Note also that if $m = 1$, then

$$(x, y) = \begin{cases} (1, -1), & \text{if } \lfloor a/2 \rfloor = \lfloor b/2 \rfloor, \\ (1, 1), & \text{otherwise,} \end{cases}$$

which is the basic idea of the plus–minus GCD algorithm of [Brent and Kung, 1983].

6.3 The new algorithm

Now let A, B be two multi-digit integers. As in the binary and plus–minus algorithms, one can skip the trailing binary zeroes in order to make A and B odd (the number of common zeroes is stored in order to be incorporated into GCD at the end of the algorithm).

Then the modular conjugates x, y of the least significant double words of A and B can be found, and the linear combination

$$C = |x * A + y * B|/2^{2m}$$

which is (roughly) one word shorter than $\max(A, B)$. Then one can make C odd by binary shifting and reiterate the procedure with C and $\min(A, B)$.

Note that if $m = 1$, then one obtains the plus–minus GCD algorithm introduced in [Brent and Kung, 1983].

However, when operating at word level, some further improvements are necessary. The inter–reduction by modular conjugates is efficient when (length(A) − length(B)) is small (for $m = 32$, we experimentally observed that the best threshold is 8). Otherwise, it is more efficient to bring the lengths closer by another scheme – for instance, by division. However, division is not suitable for parallelization, and it is also a relatively slow operation. It is better to apply instead the "exact division" scheme described in chapter 4, which works like this:

Let be $d = \text{length}(A) - \text{length}(B)$ and a, b the trailing d bits of A, B.

Set $c = (a * b^{-1}) \bmod 2^d$.

Then $C = (A - c * B)/2^d$ is (at least) d bits shorter than A.

Hence, the generalized binary algorithm consists in alternating the "exact division" step with "inter-reduction by modular conjugates" step. After each alternation, the two operands become (at least) one word shorter (typically one word is 32 bits). Note that two such steps need 3 multiplications of a simple precision integer by a multiprecision integer, compared with 4 such multiplications in Lehmer–Euclid method, but the reduction obtained is one word, while in Lehmer–Euclid only half–word reduction is achieved. This is the reason why the experimental running–time decreases by a factor of 2.

The algorithm terminates when a 0 is obtained. If 0 is obtained after an exact division step, then let $G' = B$, and if 0 is obtained after an inter-reduction step, then let $G' = (A * GCD(x, y))/y = (B * GCD(x, y))/x$. This G' will be the *approximative* GCD of initial A, B.

However, G' is in general different from $G = GCD(A, B)$, because

$$GCD(A, B) \mid GCD(B, x * A + y * B),$$

but not the other way around. A "noise" factor may be introduced at each inter-reduction step, which equals

$$GCD(B/GCD(A, B), x). \tag{6.14}$$

The combined noise must be eliminated after finding the final G' by:

$$\begin{aligned} GCD(A, B) &= GCD(G', A, B) \\ &= GCD(GCD(G', A \bmod G'), B \bmod G')). \end{aligned} \tag{6.15}$$

This "noise" is nevertheless small in the average case, because by (6.14) it is a product of some GCD's of random numbers, one from which is single precision. We experimentally observed that the length of the noise is 0.5 bits per step, in the average (in section 7.4 we compute a theoretical bound of 1.83 average bits per step). This means that G' is quite near the true GCD, hence the computations in (6.15), which have to be done by a relatively slower GCD algorithm (say Lehmer–Euclid), will only need few steps. For instance, if the difference between the input length and the GCD length is 100 words, than the (average) noise is less than two words, which needs (in average) 4 steps of Lehmer–Euclid algorithm to correct. Our experiments found that less than 5% of the GCD computation time is spent for noise elimination.

Let us also note that the computation of $x^{-1} \bmod 2^{2m}$, which is quite costly when performed via the extended Euclidean algorithm, was implemented using the scheme developed in section 4.4:

Let be $x = x_1 * \beta + x_0$. Then:

$$x^{-1} \bmod \beta^2 = (((1 - x * x') * x') + x') \bmod \beta^2,$$
$$\text{where} \quad x' = x_0^{-1} \bmod \beta$$

Using this relation, the computation of modular inverse of a double–word can be reduced to the modular inverse of a half–word, which is done by look–up in a precomputed table.

6.4 Practical experiments

The new algorithm was implemented using the GNU multiprecision arithmetic library [Granlund, 1991] and the GNU optimizing C compiler on a Digital DECstation 5000/200. For comparison purposes, the Euclidean and Lehmer–Euclid, as well as binary, plus–minus and the multiprecision versions of these were also implemented and bench marked (for more details concerning the experiments, see Chapter 8). The new algorithm runs more than 8 times faster than the Euclidean algorithm on random inputs of 100 words (32 bits per word), which is 2.5 times better than the Lehmer-Euclid scheme.

However, during practical experiments, we noticed that some of the ideas used for implementing this new algorithm can be also applied to the Lehmer multiprecision scheme. The result is an improvement of this scheme by a similar factor (see Chapter 7).

6.5 Systolic computation

The generalized binary algorithm has several characteristics which make it suitable for systolic parallelization in least–significant digits first (LSF) pipelined manner, (the particular case $m = 1$ was implemented systolically by [Brent and Kung, 1983]). All the decisions on the procedure are taken using only the lowest digits of the operands. Hence, a "master" processor computes the modular conjugates by using only the information contained in the least–significant double–words of the operands, and then sends the modular conjugates to the "slave" processors, which apply the linear combination to the rest of the digits of the operands. The cofactors and the carries can be pipelined along the string of slave processors, while the master processor can start the next cycle of the algorithm as soon as the least–significant digits of the new operand are ready.

However, there are some problems which still have to be solved in order to find a systolic version of this algorithm: noise elimination, interleaving of the two kinds of steps, shifting the operands at different rates, termination detection. We deal with these problems in chapter 11.

The LSF manner in which this algorithm operates makes it suitable for pipelined aggregation with other LSF systolic algorithms for multiprecision arithmetic (e.g. multiplication [Atrubin, 1965], exact division – see chapter 4) and with pipelined units for addition/subtraction. Hence, one has all the components required for the realization of a LSF pipelined systolic scheme for the arithmetic of multiprecision rationals.

Chapter 7

Improving the multiprecision Euclidean algorithm

We improve the implementation of Lehmer-Euclid algorithm for multiprecision integer GCD computation by partial cosequence computation on pairs of double digits, enhanced condition for exiting the partial cosequence computation, and approximative GCD computation. The combined effect of these improvements is an experimentally measured speed-up by a factor of 2 over the currently used implementation.

7.1 The multiprecision Euclidean algorithm

We present here the Euclidean algorithm and the multiprecision version of it [Lehmer, 1938] and we notice two useful properties of the cosequences. More details can be found in [Knuth, 1981, Collins, 1980]. We adopt here (with small modifications) the notations used in [Collins, 1980].

Let $A > B > 0$ be integers.

The Euclidean algorithm consists in computing the *remainder sequence* of (A, B): $(A_0, A_1, \ldots, A_n, A_{n+1})$ defined by the relations

$$A_0 = A, \quad A_1 = B, \; A_{i+2} = A_i \bmod A_{i+1}, \; A_{n+1} = 0. \tag{7.1}$$

It is well known that

$$A_n = GCD(A, B).$$

The extended Euclidean algorithm consists in computing additionally the *quotient sequence* $(Q_1, \ldots, Q_n)$ defined by

$$Q_{i+1} = \lfloor A_i / A_{i+1} \rfloor,$$

and the *first* and *second cosequences* $(U_0, U_1, \ldots, U_{n+1})$ and $(V_0, V_1, \ldots, V_{n+1})$:

$$(U_0, V_0) = (1, 0), \; (U_1, V_1) = (0, 1),$$
$$(U_{i+2}, V_{i+2}) = (U_i, V_i) - Q_{i+1}(U_{i+1}, V_{i+1}).$$

Then the following hold:

$$A_{i+2} = A_i - Q_{i+1} A_{i+1}, \tag{7.2}$$

$$A_i = AU_i + BV_i. \tag{7.3}$$

It is useful to note that the signs of the elements of each cosequence alternate:

$$\text{if } i \text{ even, then} \quad U_{i+1}, V_i \leq 0, \quad U_i, V_{i+1} \geq 0, \tag{7.4}$$

which leads to

$$(|U_{i+2}|, |V_{i+2}|) = (|U_i|, |V_i|) + Q_{i+1}(|U_{i+1}|, |V_{i+1}|). \tag{7.5}$$

These relations are useful for implementing the algorithm when the upper value of U_i, V_i is the maximum value which can be contained in a computer word (e.g. $2^{32}-1$), because then the signs are difficult to handle, since the sign-bit cannot be used.

Another useful relation is

$$|V_i| \;\geq\; Q_1|U_i|, \quad \text{for all } i \geq 1. \tag{7.6}$$

Indeed, the relation can be verified directly for $i = 1, 2$ and the induction step is:

$$|V_{i+2}| = Y_i + Q_{i+1}Y_{i+1} \geq Q_1 X_i + Q_{i+1}Q_1 X_{i+1} = Q_1|U_{i+2}|.$$

When A and B are multiprecision integers (several computer words are needed for storing the values), the divisions in (7.1) are quite time consuming. Lehmer [Lehmer, 1938] noticed that several steps of the Euclidean algorithm can be simulated by using only single precision divisions. The idea is to apply the extended algorithm to

$$a = \lfloor A/2^h \rfloor, \quad b = \lfloor B/2^h \rfloor,$$

where $h \geq 0$ is chosen such that $a > b > 0$ are single precision.

Then one gets the remainder sequence $(a_0, a_1, ..., a_k, a_{k+1})$, the quotient sequence $(q_1, \ldots, q_k)$ and the cosequences $(u_0, \ldots, u_{k+1})$ and $(v_0, \ldots, v_{k+1})$, for some $k \geq 0$ such that

$$q_i = Q_i, \quad \text{for all } i \leq k. \tag{7.7}$$

This process is called *digit partial cosequence calculation* (DPCC). When (7.7) is true, we say *the k-length quotient sequences of (a,b) and (A,B) match.* Then also

$$(u_i, v_i) = (U_i, V_i), \quad \text{for all } i \leq k+1,$$

hence by (7.3):

$$A_k = u_k A + v_k B, \;\; A_{k+1} = u_{k+1}A + v_{k+1}B, \tag{7.8}$$

and the process can be reiterated with the most significant digits of A_k, A_{k+1}.

However, the condition (7.7) cannot be tested directly, since Q_i are not known, hence one must have a condition upon a_i, q_i, u_i, v_i which ensures (7.7).

Lehmer's original algorithm computed the quotient sequences of $(a, b+1)$ and $(a+1, b)$ and compared the quotients at each step.

Collins [Collins, 1980] developed a better condition which needs only the computation of the sequences (q_i), (a_i) and (v_i). That is

$$a_{i+1} \geq |v_{i+1}| \text{ and } a_{i+1} - a_i \geq |v_{i+1} - v_i|, \text{ for all } i \leq k. \tag{7.9}$$

This condition has the advantage that only one quotient has to be computed, and only one of the cosequences. As noted by G. H. Bradley (acc. to [Knuth, 1981]), u_k, u_{k+1} can be found at the end of partial cosequence computation, by using

$$u_k a + v_k b = a_k, \ u_{k+1} a + v_{k+1} b = a_{k+1}. \tag{7.10}$$

As a comparison base for our improvements, we implemented this version of the multiprecision Euclidean algorithm using the GNU multiprecision library [Granlund, 1991] and GNU optimizing compiler on a Digital DECstation 5000/200. The benchmarks for integers with lengths varying from 5 to 100 digits (32 bit words) are shown in Fig.7.1(columns S). 1000 random integers were tested for each length. These experimental settings will remain unchanged for all the benchmarks presented in this paper.

7.2 The double digit algorithm

Recovering A_k, A_{k+1} by (7.8) involves 4 multiplications of a single digit number by a multi-digit number, and this is the most time consuming part of the whole computation. Experimentally, for 32 bit digits, one notices that the final cofactors are usually shorter than 16 bits. Hence, if digit partial cosequences computation (DPCC) would be performed on double digits, then the recovering step would require the same computational effort, but will occur (roughly) two times less frequently. This is the main idea of the improvement of Lehmer's algorithm which we will discuss in this section.

A straight forward implementation of this idea is not very efficient (see Fig.7.1): the experimental speed-up ranges from 38% (5 words operands) to 109% (100 words operands), hence is rather a slow down. However, the average values in the table show that, indeed, the number of divisions which are simulated within each step increases more than twice, which leads to a decrease of the number of steps by a factor of 2.2. The lack of speed is due to the fact that the partial cosequence computation is much slower, because of the double-digits operations involved. Therefore we will concentrate in the sequel on improving the partial cosequence computation for double words (DPCC2). Few simple improvements lead to a speed-up of almost 2:

Length	Steps per digit			Divisions per step			Time (ms)		
of operands	S	D	S/D%	S	D	S/D%	S	D	S/D%
5	1.52	0.80	190.0	7.74	17.23	44.9	0.73	1.90	38.4
25	2.28	1.03	221.4	7.58	16.94	44.7	8.25	13.30	62.0
50	2.38	1.07	222.4	7.57	16.92	44.7	25.35	31.32	80.9
75	2.41	1.08	223.1	7.57	16.93	44.7	51.17	53.03	96.5
100	2.42	1.09	222.0	7.56	16.91	44.7	85.82	78.67	109.1

Figure 7.1: Benchmarks of GCD computation using single digit DPCC (columns S) vs. double digit DPCC2 (columns D).

- (A1) Some double-digit operations can be reduced to single digit operations by using

  $$|u_i| \le |u_{i+1}| \le |v_{i+1}|, \quad |v_i| \le |v_{i+1}|,$$

  which are a consequence of (7.5) and (7.6). Therefore, only v_{i+1} has to be computed with two digits. As long as its higher digit is null, the other cofactors are also small. The speed-up ranges now from 43% to 117%.

- (A2) Digit partial cosequence computation for single words (DPCC) can be used at the beginning of DPCC2. The speed-up grows to 58% – 140%.

- (A3) Since only the second cosequence (v_i) is needed for testing the exit condition (7.9), the cofactors u_k, u_{k+1} can be computed at the end, using (7.10). However, our experiments show no improvement by this method: the speed-up decreases to 53% – 133%.

- (A4) Because the quotients are usually small (see [Knuth, 1981]), simulating division by repeated subtractions is also a source of speed-up. The idea is to repeat q_{small} times the subtraction cycle, and if the quotient is not found by then, to use division. Our experiments show that $q_{\text{small}} = 32$ is a good threshold value. The increase of the speed-up is surprisingly high: we get 101% to 180%. Note that simulating division by subtraction does not give a speed-up in the original DPCC, because the single precision division is fast enough.

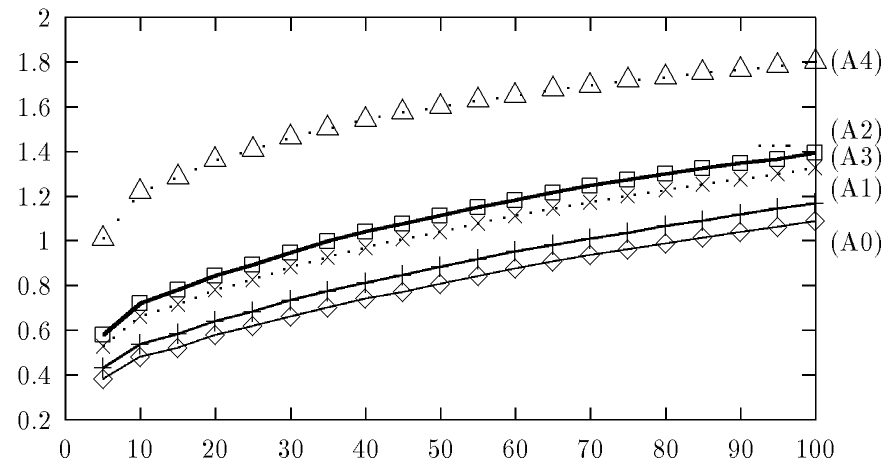

Figure 7.2: Speed-up for successive improvements of version A (length of inputs from 5 to 100 32-bit words).

The experimental results for the above variants of the algorithm are presented in Fig. 7.2, and we keep the best variant as "version A", which will be further improved in the sequel.

7.3 On the condition needed in digit partial cosequence computation

We develop here an "exact" condition for (7.7), that is, a condition which precisely indicates which is the highest k for which (7.7) is satisfied. Also we develop an "approximative" condition, which is weaker, but easier to test. We show then how to combine these two conditions, such that both efficiency and exactness are preserved.

7.3.1 An exact condition

Theorem 1. Let $a > b > 0$ be integers. Then the k-length quotient sequences of (a,b) and $(2^h b + A', 2^h a + B')$ match for any $h > 0$ and any $A', B' < 2^h$

if and only if

$$\begin{gathered} a_{i+1} \geq -u_{i+1} \quad \text{and} \quad a_i - a_{i+1} \geq v_{i+1} - v_i \quad \text{and} \quad i \text{ even} \\ \text{or} \\ a_{i+1} \geq -v_{i+1} \quad \text{and} \quad a_i - a_{i+1} \geq u_{i+1} - u_i \quad \text{and} \quad i \text{ odd} \\ \text{for all } i \leq k. \end{gathered}$$

Proof. We keep all the notations as previously introduced. Let us also denote by $(A_0, A_1, \ldots, A_{k+1})$ the elements of the remainder sequence of

$$(A, B) = (2^h a + A', \quad 2^h b + B').$$

Then

$$A_i = u_i(2^h a + A') + v_i(2^h b + B') = 2^h(u_i a + v_i b) + u_i A' + v_i B' = 2^h a_i + u_i A' + v_i B'.$$

Applying the relation

$$q = \lfloor M/N \rfloor \ \text{ iff } \ 0 \leq M - qN < N \quad (M, N, q \text{ positive integers})$$

to

$$q_i = \left\lfloor \frac{A_{i-1}}{A_i} \right\rfloor = \left\lfloor \frac{2^h a_{i-1} + u_{i-1}A' + v_{i-1}B'}{2^h a_i + u_i A' + v_i B'} \right\rfloor = \left\lfloor \frac{a_{i-1} + u_{i-1}\frac{A'}{2^h} + v_{i-1}\frac{B'}{2^h}}{a_i + u_i \frac{A'}{2^h} + v_i \frac{B'}{2^h}} \right\rfloor,$$

one gets

$$\begin{aligned} 0 \ \leq \ a_{i-1} + u_{i-1}\tfrac{A'}{2^h} + v_{i-1}\tfrac{B'}{2^h} - q_i\left(a_i + u_i\tfrac{A'}{2^h} + v_i\tfrac{B'}{2^h}\right) \ < \\ < \ a_i + u_i\tfrac{A'}{2^h} + v_i\tfrac{B'}{2^h}, \quad \forall h, A', B'. \end{aligned} \tag{7.11}$$

Using

$$u_{i+1} = u_{i-1} - q_i u_i, \ v_{i+1} = v_{i-1} - q_i v_i,$$

one notes that (7.11) is equivalent to the conjunction of

$$0 \ \leq \ \min\left\{ a_{i+1} + u_{i+1}\frac{A'}{2^h} + v_{i+1}\frac{B'}{2^h} \ \middle| \ 0 < h, \ 0 \leq A', B' < 2^h \right\},$$

$$\max\left\{ a_{i+1} - a_i + (u_{i+1} - u_i)\frac{A'}{2^h} + (v_{i+1} - v_i)\frac{B'}{2^h} \middle| 0 < h, \ 0 \leq A', B' < 2^h \right\} \ \leq \ 0,$$

The reason why the second inequality $<$ can be replaced by $\leq$ will become clear in the sequel.

Suppose i is even. Then by (7.4):

$$u_{i+1}, v_i \leq 0, \quad u_i, v_{i+1} \geq 0,$$

hence

$$\min\left\{ a_{i+1} + u_{i+1}\frac{A'}{2^h} + v_{i+1}\frac{B'}{2^h} \right\} = a_{i+1} + u_{i+1} \max\frac{A'}{2^h} + v_{i+1} \min\frac{B'}{2^h}$$

$$= a_{i+1} + u_{i+1} * 1 + v_{i+1} * 0 = a_{i+1} + u_{i+1}.$$

Also

$$\max\left\{ a_{i+1} - a_i + (u_{i+1} - u_i)\frac{A'}{2^h} + (v_{i+1} - v_i)\frac{B'}{2^h} \right\}$$

$$= a_{i+1} - a_i + (u_{i+1} - u_i) \min\frac{A'}{2^h} + (v_{i+1} - v_i) \max\frac{B'}{2^h}$$

$$= a_{i+1} - a_i + (u_{i+1} - u_i) * 0 + (v_{i+1} - v_i) * 1 = a_{i+1} - a_i + v_{i+1} - v_i.$$

Note that $B'/2^h < 1$ for any $h > 0$ and $B' < 2^h$. This is the reason why the second inequality $<$ in (7.11) can be replaced by $\leq$.

Now suppose i is odd. Similarly, one obtains

$$0 \leq a_{i+1} + v_{i+1}, \; a_{i+1} - a_i + u_{i+1} - u_i \leq 0. \square$$

The condition (7.9) can be obtained as a consequence, using (7.6).

The practical improvement of the algorithm by using this exact condition is very small. We found that the average number of divisions which are simulated in one step by using (7.9) is 16.92, vs. 17.12 by using the exact condition, hence an improvement of only 1.2% (see Fig. 7.3). However, the "exact condition" approach shows which is the maximum improvement which can be achieved by trying to increase the number of divisions per step. The speed-up over the original Lehmer's algorithm becomes 104% – 185%. We will try now to simplify the computations required for testing the condition.

Length	Steps per digit			Divisions per step			Time (ms)		
of operands	E	C	E/C%	E	C	E/C%	E	C	E/C%
5	0.80	0.80	100.0	17.42	17.23	101.1	0.70	0.72	97.2
25	1.02	1.03	99.0	17.14	16.94	101.2	5.72	5.85	97.8
50	1.06	1.07	99.1	17.13	16.92	101.2	15.48	15.83	97.8
75	1.07	1.08	99.1	17.13	16.93	101.2	29.08	29.78	97.6
100	1.07	1.09	98.2	17.11	16.91	101.2	46.50	47.63	97.6

Figure 7.3: Experiments with multiprecision GCD algorithm using "exact" condition (columns E) vs. Collins' condition (columns C).

7.3.2 An approximative condition

Note that in the context of double-digit partial cosequence computation, the condition given by Theorem 1 is not sufficient. The cofactors $u_k, v_k, u_{k+1}, v_{k+1}$ must also be smaller than a computer word (2^{32} in our case) in order to be useful. We investigate in the sequel the size of the cofactors.

For this, we use the *continuant polynomials* (see also [Knuth, 1981]) defined by

$$\begin{cases} P_0() = 1, \\ P_1(x_1) = x_1, \\ P_{i+2}(x_1, \ldots, x_{i+2}) = P_i(x_1, \ldots, x_i) + x_{i+2} * P_{i+1}(x_1, \ldots, x_{i+1}). \end{cases} \tag{7.12}$$

which are known to enjoy the symmetry

$$P_k(x_1, \ldots, x_i) = P_k(x_i, \ldots, x_1). \tag{7.13}$$

By comparing the recurrence relations (7.5) and (7.12) one notes

$$|u_i| = P_{i-2}(q_2, \ldots, q_{i-1}), \ |v_i| = P_{i-1}(q_1, \ldots, q_{i-1}). \tag{7.14}$$

Also, by transforming (7.2) into

$$a_i = a_{i+2} + q_{i+1} * a_{i+1},$$

and using $a_i > a_{i+1}$, one can prove

$$a \geq a_i * P_i(q_i, \ldots, q_1), \ b \geq a_i * P_{i-1}(q_i, \ldots, q_2).$$

Hence by (7.13) and (7.14) one has

$$|v_{i+1}| \leq a/a_i, \ |u_{i+1}| \leq b/a_i.$$

Since a_0, a_1 are double digits, we have

$$2^{2n-1} \leq a = a_0 < 2^{2n}, \quad b = a_1 < 2^{2n},$$

where n is the bit-length of the computer word (in our implementation $n = 32$). Suppose

$$a_{i+1} > a_{i+2} \geq 2^n.$$

Then

$$|v_{i+1}| \leq a_0/a_i < a_0/2^n < 2^n < a_{i+1},$$

(note that $a_{i+1} \geq 2^n$ is sufficient here).

Also

$$a_i - a_{i+1} \geq a_i - q_{i+1} * a_{i+1} = a_{i+2} \geq 2^n,$$

$$|v_i| + |v_{i+1}| \leq |v_i| + q_{i+1} * |v_{i+1}| = |v_{i+2}| \leq a/a_{i+1} < a/2^n < 2^n.$$

The previous relations allow us to develop an alternative to (7.9), which is

$$a_{i+2} \geq 2^n. \tag{7.15}$$

For the practical implementation, this means that the high order digit of a_{i+2} is not zero, which is computationally inexpensive to test.

Also, note that when (7.15) holds, then

$$|v_i| \leq |v_{i+1}| \leq |v_{i+2}| \leq |v_{i+3}| \leq a/a_{i+2} < 2^n,$$

$$|u_i| \leq |u_{i+1}| \leq |u_{i+2}| \leq |u_{i+3}| \leq b/a_{i+2} < 2^n,$$

hence the sizes of the cofactors are "good" for $k = i, \ k = i+1, \ k = i+2$. Using this, we combine the "exact" and the "approximative" condition in order to obtain maximum of efficiency:

- In the main loop of DPCC2, the maximum i for which (7.15) holds is determined, and then we know $k = i$ is "good".
- After exiting the loop, we test the only second part of the condition (because $a_{i+1} \geq 2^n$ implies the first part), and if this test passes then we know $k = i + 1$ is "good".

Length	Steps per digit			Divisions per step			Time (ms)		
of operands	B0	B1	B0/B1%	B0	B1	B0/B1%	B0	B1	B0/B1%
5	0.80	0.80	100.0	17.42	17.41	100.1	0.70	0.63	111.1
25	1.02	1.02	100.0	17.14	17.13	100.1	5.72	5.30	107.9
50	1.06	1.06	100.0	17.13	17.11	100.1	15.48	14.58	106.2
75	1.07	1.07	100.0	17.13	17.12	100.1	29.08	27.80	104.6
100	1.07	1.08	99.1	17.11	17.09	100.1	46.50	44.92	103.5

Figure 7.4: Benchmarks of version B (improved condition in DPCC): "exact" condition (columns B0) vs. "combined" condition (columns B1).

- Full condition must be tested in order to insure $k = i + 2$ is "good".

According to our experiments, if $a_{i+2} \geq 2^n$ and $a_{i+3} < 2^n$, then the maximum "good" value of k is $k = i$ in about 10% of the cases, $k = i + 1$ in 20% of the cases, and $k = i + 2$ in 70% of the cases. The incidence of situations when $k = i + 3$ is "good" is insignificant.

In Fig.7.4 we present the comparative benchmarks for the algorithm using exact condition (columns B0) and approximative combined with exact (columns B1). One can see that the number of simulated divisions per step decreases insignificantly, but the running time improves: the speed-up is now 116% – 191% (see Fig.7.5). We keep the best variant as "version B", which will be further improved in the sequel.

7.4 Approximative GCD computation

The final improvement is based on the following idea: instead of computation (7.8), which requires 4 long multiplications, compute only A_{k+1} and continue the algorithm with B and A_{k+1}. Then, since A_{k+1} is about one word shorter than B, a division will be performed at the next step, which is computationally equivalent to one long multiplication. Hence, one step of Lehmer's scheme (4 long multiplications) will be replaced by one "half–step" and one division (3 long multiplications), reducing the number of digit multiplications by 25%.

However, this new scheme lacks in correctness, since $GCD(A, B)$ divides $GCD(B, A_{k+1})$, but they might be different. Hence the final result G' of the

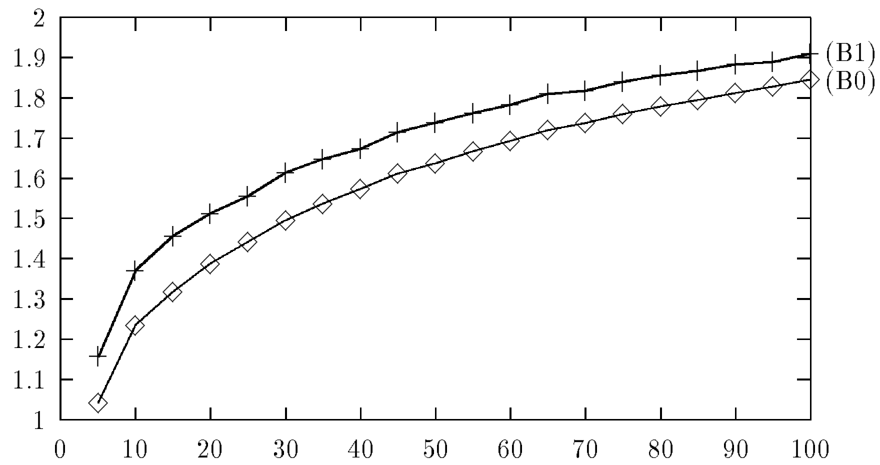

Figure 7.5: Speed-up for successive improvements of version B (length of inputs from 5 to 100 32-bit words).

algorithm will be, in general, bigger than the real GCD G, but still divisible by G. Then G' can be used in finding G by:

$$GCD(A, B) = GCD(A, B, G') = GCD(A \bmod G', GCD(B \bmod G', G')).$$

If the "approximative" GCD G' is "near" the real GCD G, then these two GCD computations will be inexpensive.

One can see that the "noise" $GCD(B, A_{k+1})/GCD(A, B)$ introduced at each step equals $g = GCD(u_{k+1}, B/G)$, because u_{k+1}, v_{k+1} are relatively prime. If one assumes u_{k+1} and B/G independently random, then the probability that the noise g at one step is m bits long is:

$$p_m = \sum \left\{ \frac{p'_i}{i^2} \;\middle|\; 2^{m-1} \le i \le 2^m - 1 \right\},$$

where p'_i is the probability that u_{k+1}/i and $(B/G)/i$ are prime: $p'_i \le 1$. Hence $p_m \le s_m$, where

$$s_m = \sum \left\{ \frac{1}{i^2} \;\middle|\; 2^{m-1} \le i \le 2^m - 1 \right\} = \frac{\Psi(1, 2^m) - \Psi(1, 2^{m-1})}{4}$$

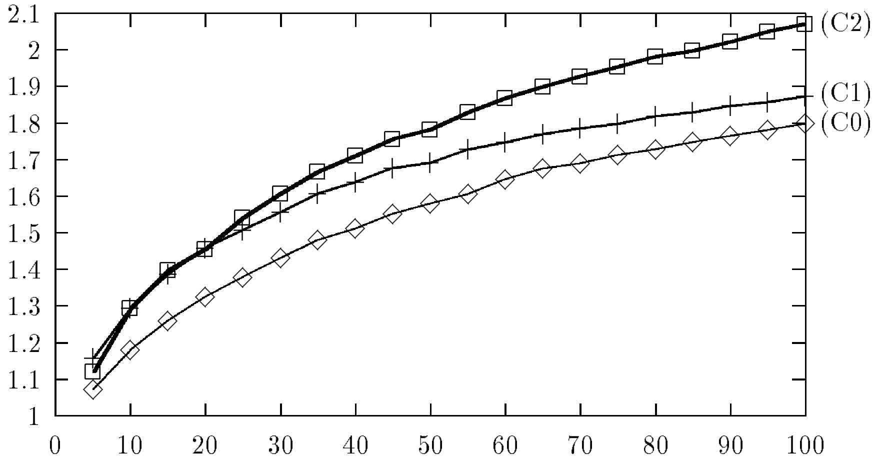

Figure 7.6: Speed-up for successive improvements of version C (length of inputs from 5 to 100 32-bit words).

using `maple` system, and furthermore:

$$\sum_{m=2}^{32} ms_m \approx 1.83,$$

which is an upper bound for the the average bit-length of the noise per step. This bound seems to be very coarse, since experimentally we detected a noise of 0.52 bits per step. This means 100 steps will give an average noise of less than two digits, whose correction requires only four steps of the original Lehmer-Euclid scheme. Fig.7.6 presents the speed-up obtained for the "approximative" GCD computation.

(C0) The straight forward implementation of the idea does not give a satisfactory result (the speed-up decreases to 107% – 180%), because in the original program a division is performed only when the length-difference of A, B exceeds one word. The experiments show that this happens only in about 7% of the cases, hence there is no alternation between Lehmer–type and division steps.

(C1) By setting the threshold for testing the length–difference at 24 bits, the percentage of the division steps becomes 49%, hence the desired alternation is realized. The speed–up improves a little (116% – 189%), but it is still not satisfactory, probably because of the additional time consumed for calling the division routine.

(C2) The final variant is obtained by replacing the calls to the division routine $A \bmod B$ (in cases when the length–difference is at most one word), with an explicit computation of $A - q * B$, where q is the quotient of the most-significant double-digits of A and B, which has the property

$$q - 1 \leq \lfloor A/B \rfloor \leq q,$$

(see [Knuth, 1981]). Hence $|A - q * B|$ equals either $(A \bmod B)$ or $(A - A \bmod B)$, which are both good for continuing the GCD computation. This subvariant performs a little better than the previous one (speed-up 112% – 207%), it gives some improvement over version B, and also, for large lengths (80 words), overcomes the "psychological barrier" of two times speed–up in comparison with the original Lehmer algorithm. This last variant is kept as "version C" of the algorithm.

Fig.7.7 presents the speed–up for the 3 main versions. The absolute timings (on DECstation 5000/200) range from 0.65 ms to 41.43 ms for the improved algorithm, versus 0.73 to 85.82 ms for the original Lehmer-Euclid algorithm.

In fact, the figure shows an increase of the speed-up for bigger inputs. We obtained 230% speed-up for 300 word operands on a DECstation 5000/240 (see Fig.7.8).

7.5 Conclusions

Starting from the idea of using double digits in digit partial cosequence computation, and applying various improvements, one can speed-up the multiprecision Lehmer-Euclid algorithm by a factor of 2.

Although some of the improvements are related to the particular hardware used for experiments (DECstation 5000/200, DECstation 5000/240), most of the ideas will work for any sequential machine (a correction of thresholds might be necessary).

Until recently, the Lehmer-Euclid algorithm was considered most efficient for practical applications. However, new research on improving the binary algorithm of [Stein, 1967] (see [Sorenson, 1994, Weber, 1993], chapter 6, chapter 8) found faster GCD algorithms. The improved Lehmer-Euclid method presented here is in the same range as these ones.

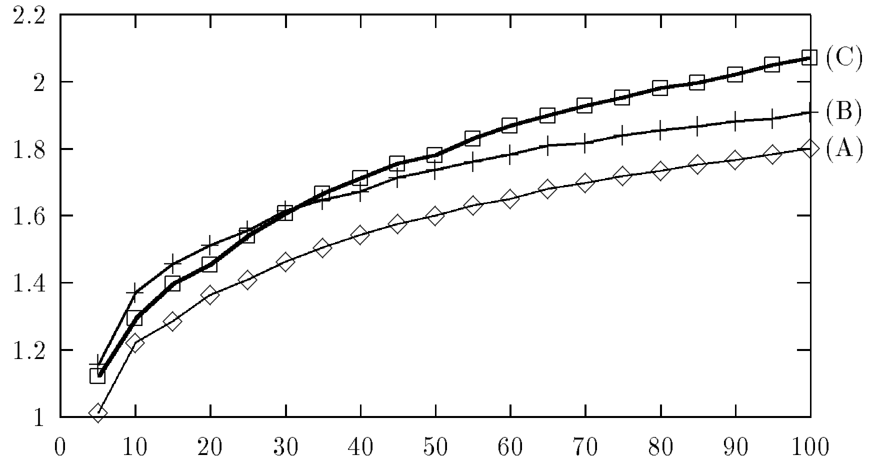

Figure 7.7: Speed-up for the three main versions, depending on the length of inputs (in 32-bit words).

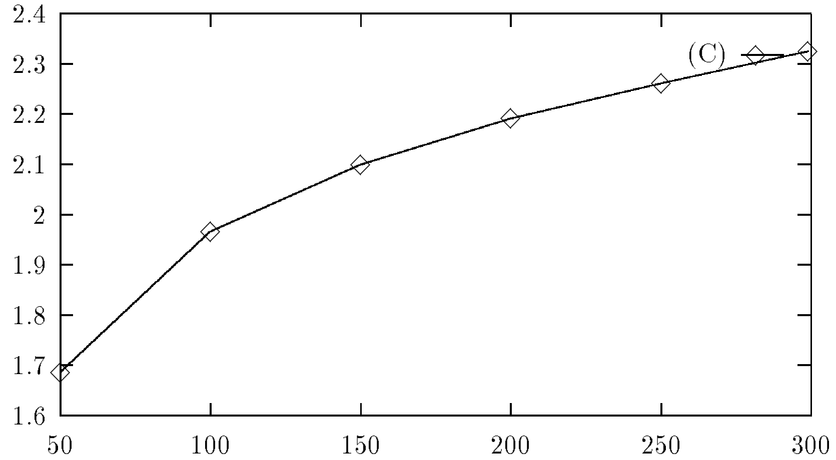

Figure 7.8: Speed-up for version (C) on very long inputs (50 to 300 words of 32 bits).

A natural extension of this double-digit Lehmer-Euclid would be the extended Euclidean algorithm. Currently [Collins and Encarnación, 1994] is improving the extended Euclidean algorithm and the multiprecision integer-to-rational algorithm of [Wang *et al.*, 1982] by using Lehmer-like approach and double-digit DPCC.

Chapter 8

Comparing several GCD algorithms

We compare the execution times of several algorithms for computing the GCD of arbitrary precision integers. These algorithms are the known ones (Euclidean, binary, plus-minus), and the improved variants of these for multi-digit computation (Lehmer and similar), as well as new algorithms introduced by the author: an improved Lehmer algorithm using two digits in partial cosequence computation, and a generalization of the binary algorithm using a new concept of "modular conjugates". The last two algorithms prove to be the fastest of all, giving a speed-up of 7 to 8 times over the classical Euclidean scheme, and 2 times over the best currently known algorithms.

8.1 Introduction

We report here on the computing time of multiprecision GCD computation by the following algorithms:

- **Euclid**, **I-Euclid**, **I-I-Euclid**: The classical Euclidean scheme, the improvement of it for multidigit integers ([Lehmer, 1938]), and a further improvement by the author using two digits in partial cosequence computation – see chapter 7.
- **binary**, **I-binary**: The binary GCD algorithm ([Stein, 1967]) and its improvement for multidigit integers (Gosper, see [Knuth, 1981]).
- **PlusMinus**, **I-PlusMinus**: The plus-minus scheme introduced in [Brent and Kung, 1983] and its improvement for multidigit computation.
- **G-binary**, **I-G-binary**: A new algorithm for multiprecision GCD, which generalizes the binary and plus-minus GCD algorithms and its improvement by using two digits in computation of cofactors – see chapter 6.

We did not consider the GCD algorithms based on FFT multiplication scheme ([Schönhage, 1971], [Moenck, 1973b]), which are asymptotically faster, but are not expected to give a practical speed-up for the range of integers we are interested in (up to 100 words of 32 bits).

The results of the experiment show the following speed-up over the raw Euclidean scheme, for random pairs of integers with 100 words of 32 bits:

- 2 times for **binary** and **PlusMinus**;
- 3.5 times for **I-Euclid** and **I-PlusMinus**;
- 5 times for **I-binary** and **G-binary**;
- 7.5 times for **I-I-Euclid**;
- 8 times for **I-G-binary**.

8.2 Description of the algorithms

We present here the outline of the GCD algorithms which were measured, and indicate the appropriate literature for the readers which are interested in more details concerning the correctness proofs and complexity analysis.

8.2.1 Euclid

Starting with two positive integers $A_0 \geq A_1$, one computes the remainder sequence $\{A_k\}_{1 \leq k \leq n+1}$ defined by the relations:

$$A_{k+2} = A_k \bmod A_{k+1}, \quad A_{n+1} = 0.$$

and then one has: $GCD(A_0, A_1) = A_n$.

An extensive discussion on Euclidean algorithm is presented in [Knuth, 1981].

8.2.2 I-Euclid

The *extended* Euclidean algorithm (also in [Knuth, 1981]) consists in computing the additional sequences $\{q_k\}_{1 \leq k \leq n}, \{u_k, v_k\}_{0 \leq k \leq n+1}$ defined by:

$$\begin{gathered} q_{k+1} = \lfloor A_k / A_{k+1} \rfloor, \\ u_0 = 1, v_0 = 0, u_1 = 0, v_1 = 1, \\ u_{k+2} = u_k + q_{k+1} * u_{k+1}, \\ v_{k+2} = v_k + q_{k+1} * v_{k+1}. \end{gathered} \tag{8.1}$$

which have the properties:

$$A_k = \begin{cases} u_k * A_0 - v_k * A_1, & \text{if } k \text{ even}, \\ -u_k * A_0 + v_k * A_1, & \text{if } k \text{ odd}, \end{cases} \tag{8.2}$$

The sequences $\{u_k, v_k\}$ of *cofactors* are called *cosequences* of $\{A_k\}$.

If A_0 and A_1 are multiprecision integers, then let a_0 and a_1 be the most significant 32 bits of A_0 and the corresponding bits of A_1. Lehmer [Lehmer, 1938] noticed that part of the sequence $\{q_k\}$ can be computed by:

$$q'_{k+1} = \lfloor a_k / a_{k+1} \rfloor, \quad a_{k+2} = a_k \bmod a_{k+1}, \tag{8.3}$$

and as long as $q'_k = q_k$, the sequences $\{u_k, v_k\}$ computed as in (8.1) are the correct ones. Hence, the algorithm **I-Euclid** simulates several steps of the Euclidean algorithm by using only simple precision arithmetic. This process is called *digit partial cosequence computation* (see [Collins, 1980]). When $q'_{k+2} \neq q_{k+2}$, then A_k and A_{k+1} are recovered using (8.2), and the cycle can start again.

The algorithm needs a sufficient condition for $q'_{k+1} = q_k$. We have used:

$$a_{k+1} \geq v_{k+1} \quad \textbf{and} \quad (a_k - a_{k+1}) \geq (v_k + v_{k+1}), \tag{8.4}$$

which was developed in [Collins, 1980].

8.2.3 I-I-Euclid

In **I-Euclid**, recovering A_k, A_{k+1} involves 4 multiplications of a single digit number by a multidigit number, and this is the most time consuming part of the whole computation. Experimentally, one notices that final cofactors are usually shorter than 16 bits. Therefore, if partial cosequences would be computed for pairs of double digits, then the recovering step would require the same computational effort, but will occur (roughly) two times less frequently. This idea suggests the next improvement of the Euclidean algorithm.

I-I-Euclid (described in detail in chapter 7) uses *double-words* in cosequence computation. The number of double-words operations is reduced to the minimum necessary by using some properties of the cosequence and a new *termination condition* for the inner loop. The speed is also improved by using *approximative* computation: only A_k is recovered after a Lehmer-loop, hence the result obtained will have to be corrected in order to get the true GCD.

8.2.4 Binary and PlusMinus

The **binary** GCD algorithm ([Stein, 1967], [Knuth, 1981], [Brent, 1976]) is based on the relations:

$$\begin{aligned} &GCD(A,B) = GCD(A-B,B),\\ &\text{If } A \text{ odd, } B \text{ even, then}\\ &\qquad GCD(A,B) = GCD(A,B/2). \end{aligned} \tag{8.5}$$

The algorithm begins by shifting A_0, A_1 rightwise as many positions as there are zero trailing bits, and stores the number of common zero bits for being incorporated into the resulting GCD at end of computation. After shifting A_0, A_1 are odd, hence $A_2 = |A_0 - A_1|$ is even. A_2 is then shifted rightwise for skipping the zero bits and the cycle is repeated with A_2 and $\min(A_0, A_1)$. Note that it is necessary to know at each step which of A_k, A_{k+1} is the largest. When $A_k = 0$, then the GCD is A_{k-1} shifted leftward with the number of common zero bits of A_0, A_1.

The multidigit version **I-binary** (see [Knuth, 1981]) is developed from **binary** in the same way **I-Euclid** is developed from **Euclid**. For a certain number of steps, one can decide which shifts/subtractions are needed by only looking at the most–significant and least–significant digits of A_0, A_1. These shifts and subtractions are encoded into cofactors which allow the recovery of A_k, A_{k+1} when single digit operation is not possible anymore.

The need to compare A_k, A_{k+1} at each step prevents efficient parallelization of **binary**. In order to overcome this, the **PlusMinus** algorithm was developed in [Brent and Kung, 1983]. This algorithm is based on (8.5) and supplementary relations:

$$\begin{aligned}&GCD(A, B) = GCD(A + B, B),\\ &\text{If } A, B \text{ odd, then } 4|(A + B) \textbf{ or } 4|(A - B).\end{aligned}$$

Therefore, each step consists of choosing $A'_{k+2} = A_k + A_{k+1}$ or $A'_{k+2} = A_k - A_{k+1}$ such that $4|A'_{k+2}$, and then setting A_{k+2} to A'_{k+2} shifted rightward to skip the zero bits. Note that in this case it is not important anymore to know which of A_k, A_{k+1} is the largest. The scheme works even if one (or both) of the operands become negative.

This makes easier the implementation of **I-PlusMinus**, since there is no need to keep track of the most–significant digits of A_0, A_1.

Note also that further improvement of **I-binary** and **I-PlusMinus** is not possible in the way we did it for **I-Euclid**, because in this case the cofactors are not bound by half-word size.

8.2.5 G-Binary

This algorithm is described in detail in chapter 6 and it is based on a natural generalization of the plus-minus scheme:

> Let $m \geq 1$ be a constant. Given positive long integers A, B, let a, b be the least significant $2m$ bits. Find x, y with at most m bits, such that:

$$2^{2m} \mid (x * a \pm y * b) \tag{8.6}$$

The *modular conjugates* x, y of a, b can be found by applying the extended Euclidean scheme to 2^{2m} and $(a * b^{-1}) \bmod 2^{2m}$.

Note that for $m = 1$ the plus-minus scheme is obtained.

Now let A, B be two multidigit odd integers. By applying the scheme above we get $C = (x * A \pm y * B)/2^{2m}$ which is (roughly) m bits shorter than $\max(A, B)$. This is efficient when $(\text{length}(A) - \text{length}(B))$ is small (for $m = 64$, we experimentally observed that the best threshold is 8). Otherwise, it is more efficient to bring the lengths closer by the "exact division" scheme described in chapter 4, which works like this:

Let be $d = \text{length}(A) - \text{length}(B)$ and a, b the trailing d bits of A, B.

Set $c = (a * b^{-1}) \bmod 2^d$.

Then $C = (A - c * B)/2^d$ is (roughly) d bits shorter than A.

Hence, the generalized binary algorithm consists in alternating the "exact division" step with "inter-reduction by modular conjugates" step. After each alternation, the two operands become (roughly) m bits shorter ($m = 16$ for **G-binary** and $m = 32$ for **I-G-binary**).

This computation is also approximative (as in **I-I-Euclid**), hence a correction step must be applied at the end.

8.3 Experiment settings and results

We implemented the algorithms using the GNU multiprecision arithmetic library [Granlund, 1991], under the GNU optimizing C compiler. The experiments were done using a the Digital DECstation 5000/200 (MIPS RISC architecture, 25 MHz, 24 MIPS). For each length, each of the algorithms was applied to 1000 pairs of random integers.

The figures 8.3 and 8.3 present the absolute timings in milliseconds and the speed-up over the raw Euclidean algorithm (**Euclid**). The absolute timings for 100 word operands are 317 milliseconds for the raw Euclidean algorithm and 39 milliseconds for improved generalized binary (**I-G-binary**).

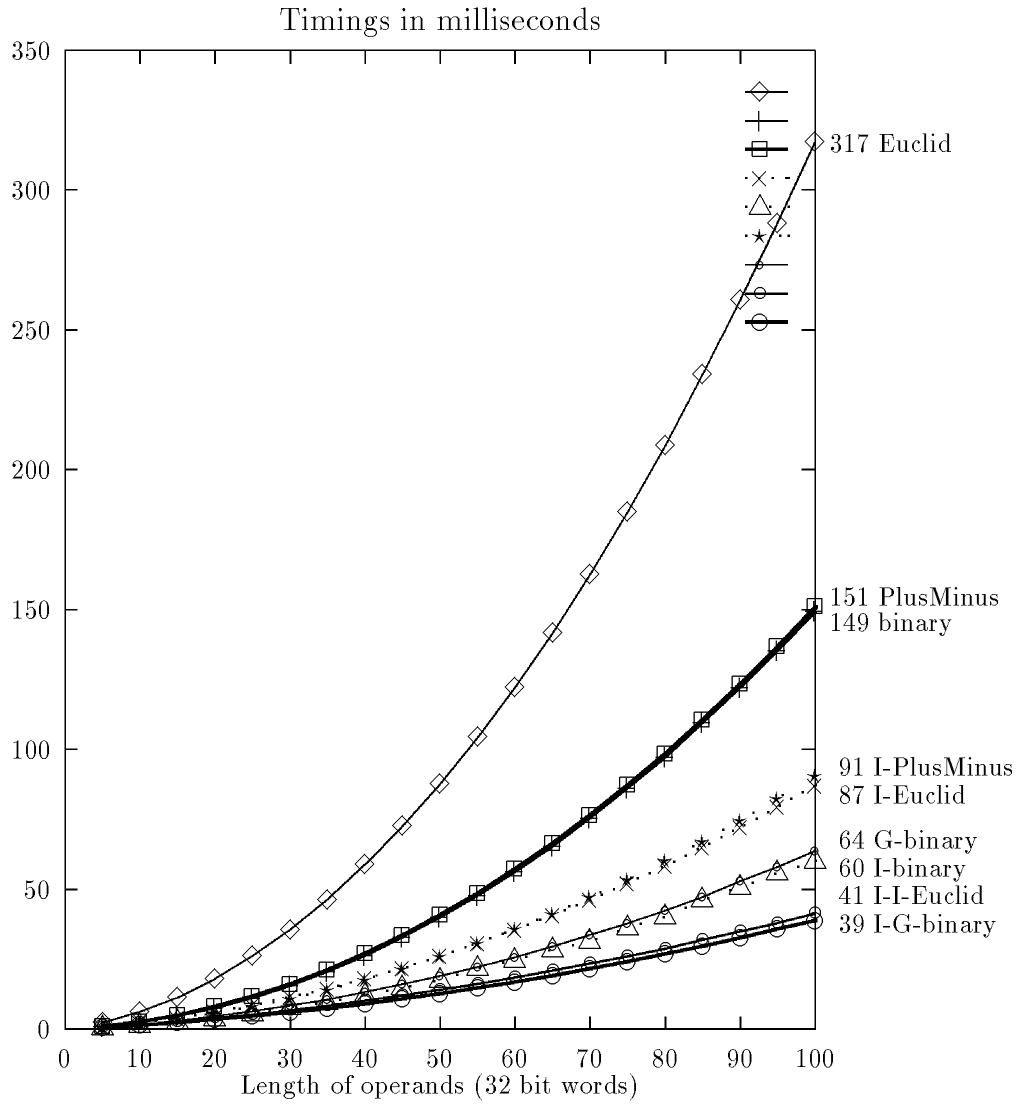

Figure 8.1: Comparison of absolute timings.

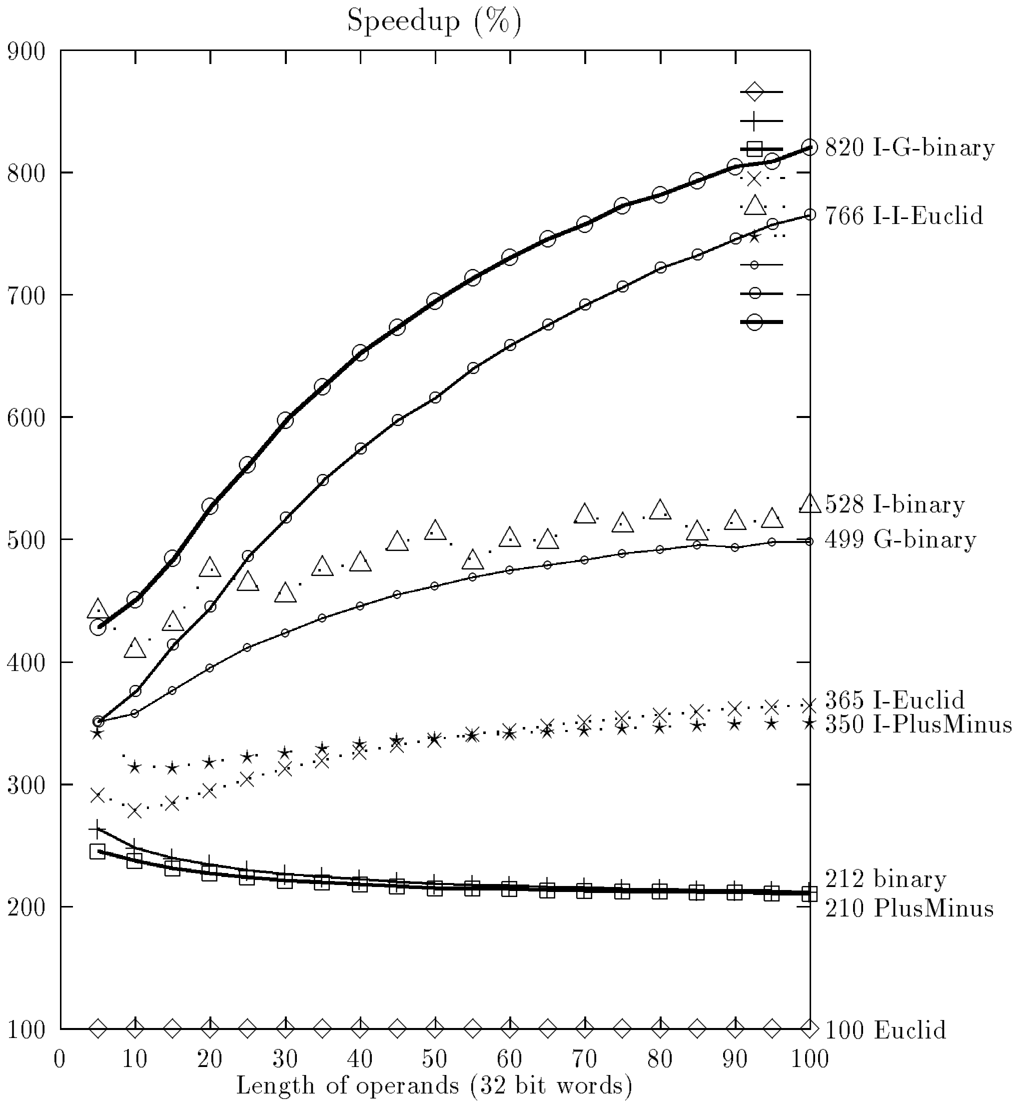

Figure 8.2: Speed-up over the Euclidean algorithm.

Chapter 9

Implementation: Which architecture?

We review some parallel architectures which are available within the ACPC network and elsewhere, and we evaluate them with respect to the possibility of efficient implementation of fine grained systolic algorithms for the use of Computer Algebra systems.

The conclusion is that the devices which are more likely to constitute an appropriate hardware environment for such an attempt are the Field Programmable Gate Arrays. Also, the SIMD architecture of MasPar offers a good testing environment.

9.1 Overview of the investigation

Of course, the most efficient implementation could be probably obtained only by designing and effectively realizing one or several dedicated chips, which is possible by using currently available CAD systems and the cooperation of an Electronics Engineering Department and a Microprocessor Company. However, this approach has several major disadvantages:

- The turn–around time of such implementations is several months in the best case, which could lead to 1 to 3 years of development time.
- The expenses involved are much bigger than those required by software implementation.
- The knowledge and skills required in the actual implementation phase are only remotely related to our expertize.
- At the moment, several big companies are investing efforts far beyond our means in designing very powerful chips. It is quite likely that the results they will obtain in the next few years, even by using the conventional sequential approach, will surpass our result.

Therefore, we believe that a more realistic approach is to try to use the hardware currently offered by the leading companies in the field, in such a way that the results of our research could be easily implemented on the future versions of this hardware.

In investigating the currently available parallel hardware, one could be easily overwhelmed by the large diversity of architectures which have been recently released or are under development. Therefore, we choose a sampling approach, by first investigating the hardware which is currently available in RISC–Linz or in the ACPC network, and then some representative systolic (iWarp) and cellular automata (CAM–PC) computers, and Field Programmable Gate Arrays with systolic architecture (CAL1024, CLi6000). This sampling is already sufficient for giving us the relevant information for concluding this investigation.

The following parallel hardware is currently available within ACPC network:

- Sequent Symmetry (RISC–Linz)
- Impuls Multi–Transputer (RISC–Linz)

- MasPar MP–1 (TU Vienna)
- N–Cube (TU Vienna)

We do not discuss here the N–Cube, which is not suitable for our application.

The table in Fig.9.1 summarizes the technical data, which is discussed in more detail in the next section.

9.2 Evaluation details

In this section we review the technical data which is relevant for our approach, and we evaluate the devices with respect to the possibility of efficiently implementing fine grain parallelism based on the systolic approach.

9.2.1 Sequent Symmetry

- **Technical Data:**
  - Type: MIMD shared memory.
  - Processors, connection: 20 PE (processing elements), global bus
  - Processor speed: 1.5 MIPS (20 MHz clock).
  - Communication speed, capacity: 4 MB/sec, 8–16 bits.
  - Operating system, programming: Unix like, C, μSystem, paclib.
- **Evaluation:** Fine grain parallelism is not suitable for this machine, because of parallelization/communication overhead.

  The global bus connection is not suitable for systolic communication.

  The operating system and the hardware support the implementation of large Computer Algebra systems.
- **Conclusion:** The systolic approach can be used for a coarse level parallelization, or by simulating several cells on each processor. However, these two approaches are only remotely related to our purpose.

9.2.2 Impuls Multi–Transputer(T800)

- **Technical Data:**
  - Type: MIMD distributed memory, driven by a PC host.

Name	Structure	PE speed	Communic.	Programming	Price
		Parallel Computers			
Sequent Symmetry	MIMD shared mem 20 PE global bus	20 MHz 1.5 MIPS	4 MB/s (8–16 bits)	C μSystem **pac**	Available RISC
Impuls Multi-Transputer (T800)	MIMD 16 PE 4 crossbar sw.	17.5 MHz 15 MIPS	2.5 MB/s (1 bit)	C(Logical Systems)	Available RISC
MasPar MP–1	SIMD 32x32 PE Xnet	14 MHz 1.6 MIPS	2.5 MB/s (1 bit)	C(MPL)	Available ACPC
iWarp (Intel)	MIMD 32x32 PE +net	20 MHz 20 MIPS	40 MB/s (4 bits)	C (iWarp)	$10,000/PE $300,000(32 PE)
		Cellular Automata			
CAM–PC (Automatrix)	256x256 cells (4 bits) Xnet	60 Hz	neighbors direct access (16 bits)	CAM–FORTH (64KB look-up)	$2,000/system
		Field Programmable Gate Arrays			
CAL–1024 (Algotronix)	32x32 cells (1 bit) Xnet	125MHz	neighbors direct access (1 bit)	CAD software (host memory)	$500/chip(32x32) $32,000 (64 chips) $10,000/system
CLi6000 (Concurrent Logic)	80x80 cells (1 bit) +net,buses	200–1000MHz	neighbors direct access (2 bit)	CAD software (loaded)	$350/chip(56x56) $15,000(40 chips) $17,000/system

Figure 9.1: Summary of technical data

  - Processors, connection: 16 PE, 4 crossbar switches allowing almost arbitrary configuration.
  - Processor speed: 7.5 MIPS (17.5 MHz).
  - Communication speed, capacity: 2.5 MB/sec, 1 bit serial.
  - Operating system, programming: MS–DOS, C (Logical Systems).

- **Evaluation:** Fine grained parallelism is not suitable for this machine, because of the low number of processors. However simulation of only 5 cells/processor could give interesting results for our research.

  The crossbar switch connection allows the implementation of systolic communication with no significant slow–down.

  The operating system and the machine do not support standard large Computer Algebra systems.

- **Conclusion:** This is a useful tool for testing the fine grained systolic approach on a MIMD machine, but it does not offer all the characteristics needed by our approach.

9.2.3 MasPar MP–1

- **Technical Data:**
  - Type: SIMD distributed memory, driven by a VAXstation host.
  - Processors, connection: 1024 PE, 32 by 32 mesh, 8 neighbors connection (Xnet), global broadcasting from an Array Control Unit.
  - Processor speed: 1.6 MIPS (16 MHz)
  - Communication speed, capacity: 2.5 MB/sec, 1 bit serial.
  - Operating system, programming: Unix like, C (MPL).

- **Evaluation:** Fine grain parallelism and systolic communication are suitable, the SIMD architecture eases the synchronization.

  However, the same SIMD architecture restricts the full parallelism achievable in our application: it is impossible to perform in parallel different tasks on different processors, or to put/get data into/from the host in parallel with the array processing.

  The operating system and hardware of the host support large Computer Algebra systems.

- **Conclusion:** This is a powerful tool for testing the fine grain systolic approach on a SIMD machine, but it does not offer all the characteristics needed by our approach.

9.2.4 iWarp

- **Technical Data:**
  - Type: MIMD distributed memory (based on the systolic approach), driven by a Sun host.
  - Processors, connection: up to 1024 PE, 32 by 32 (or others) mesh, 4 neighbors connection (+net).
  - Processor speed: 20 MIPS (20 MHz).
  - Communication speed, capacity: 40 MB/sec, 4 bits, special hardware for systolic communication.
  - Operating system, programming: Unix, C (iWarp).
  - Price: Depending on configuration, roughly \$10.000/processor: \$40.000 (4 PE – minimal configuration), \$300.000 (32 PE – minimal configuration interesting for our research), \$1.94 million (256 PE), \$7.75 million (1024 PE).
  - Producer: Intel
- **Evaluation:** iWarp is probably **the** systolic computer on the market, and it meets all the requirements of fine grain systolic implementations. Also, systolic implementation of algebraic algorithms at various levels of granularity can be performed on this machine.

  However, the ratio price/performance would be quite high for our application. For instance, a good implementation of a systolic long integer multiplication algorithm would yield a speed up of 50 to 100 over the sequential one, for 32 words operands. However, this algorithm would make very poor use of the resources available in a 32 node iWarp (in each cell: only few registers, the integer arithmetic unit, and the systolic communication hardware would be used).

  The operating system and hardware of the host support large Computer Algebra systems.
- **Conclusion:** This would be a good tool for testing/implementing fine grain systolic algorithms, but only when the ratio price/performance is not considered.

9.2.5 CAM–PC

- **Technical Data:**
  - Type: Cellular Automaton, driven by a PC host.
  - Processors, connection: 65,536 4 bits cells per board, 256 by 256 mesh, Xnet, up–scaling possibilities by horizontal and vertical gluing of several boards.
  - Processor speed: 60 Hz (60 array updates per second) (!).
  - Communication speed, capacity: 16 bits at each step, by direct access between neighbors.
  - Operating system, programming: MS–DOS, CAM–PC FORTH, C(beta release).
  - Price: $2,000 basic package, $6,000 including PC host.
  - Producer: Automatrix, Inc. (Rexford, New York).
- **Evaluation:** Suitable for fine grain systolic algorithms.

  However, the very low speed of the device makes such an attempt only of an academic interest.

  The operating system and the host do not support standard large CA systems.
- **Conclusion:** Useful for testing in a pure cellular automata environment, but not interesting for our purpose because of the low speed.

9.2.6 CAL1024

- **Technical Data:**
  - Type: SRAM Field Programmable Gate Array with systolic architecture, driven by PC or Sun host.
  - Processors, connection: 1024 cells per chip, 32 by 32 mesh, +net, scalable by gluing. Each cell can be programmed to perform one of the 16 logic function with 2 bits input and one bit output, or to act as a 1 bit register. Simultaneously, each cell can perform routing between its neighbors. The chips are seen as SRAM memory by the host, hence all data is easy accessible, and partial and dynamic reconfiguration is possible.

  - Processor speed: 8 nsec delay per computing cell (125MHz). The clock speed depends on the application (longest computation path in one step). For instance, if this is 6, at most 20MHz are possible. A user has programmed a self-timed FIFO array which works at 15MHz.
  - Communication speed, capacity: Practically no delay between neighbors, 1.6 ns delay for each transit cell.
  - Operating system, programming: MS–DOS or Unix, C, custom CAD software.
  - Price: $500 per chip. Minimal system configurations (including software) are: $3,000 for 2 CAL chips and PC software, $10,000 for 8 CAL chips and Sun software.
  - Producer: Algotronix Ltd, Edinburgh, UK.

- **Evaluation:** A good environment for testing and implementing fine grain systolic algorithms. It offers the flexibility of customized hardware, but without the disadvantages of the actual chip manufacturing process. This kind of hardware supports the implementation of the "active memory" computation paradigm, in which the memory itself is used as a computation agent. This new paradigm is likely to increase dramatically the hardware usage and to decrease the ratio price/performance for intensively computation tasks.

  The speed would probably be lower that the one of a fully custom solution, but still in the range of giving a significant speed–up for our application.

  In Sun version, the host supports large CA systems.

  A faster and denser CAL chip, although structurally compatible with the old one, will be available next year.

- **Conclusion:** Suitable for or purpose.

9.2.7 CLi6000

- **Technical Data:**

  - Type: SRAM Field Programmable Gate Array with systolic architecture, driven by PC host.

  - Processors, connection: 576 to 6400 cells per chip, 24 by 24 to 80 by 80 mesh, +net, buses for distant connections; scalable by horizontal gluing. Each cell can be programmed to perform a 2 to 2 bits function from a selected set, or to act as 1 bit register, possibly combined with some multiplexing function.
  - Processor speed: 1 to 5 nsec delay per cell (200 to 1000MHz). Same as for CAL, the clock speed depends on the application. For instance, a 4 bit ripple carry counter runs at 90MHz.
  - Communication speed, capacity: no delay between adjacent cells, 1 to 5 nsec delay on buses.
  - Operating system, programming: MS–DOS, customized CAD software. The final target program (string of bits) is loaded into the chip by the CAD software. Partial and dynamic reconfiguration is possible.
  - Price: Ranges from $90 (32 by 32 chip) to $350 (56 by 56 chip). A minimal configured system including CAD software is $6,500, a configuration including all software tools (optimization, verification, debugging) is $17,000.
  - Producer: Concurrent Logic, Inc. (Sunnyvale, California), with sales representative in Germany.

- **Evaluation:** Most of the evaluation remarks about CAL1024 are also valid here. It is more relevant to make a comparative evaluation of the two devices:

  - **Advantages of CLi6000 over CAL1024**
    - More computing power is embedded in each cell. For instance, a 4 bit counter needs 8 CLi cells vs. 24 CAL cells.
    - The buses for long distance communication allow easier and less area consuming implementations. (For instance, in the CAL implementation of the 4 bit counter, 15% of the cells are used for communication.)
    - More cells are available for a lower price: $350/3000 CLi cells vs. $500/1000 CAL cells, that is $0.12/cell vs. $0.50/cell.
    - CLi is 2 to 3 times faster.
  - **Disadvantages of CLi6000 w. r. t. CAL1024**

  * CAL is "higher level": the logic performed by a CAL cell is selected out of the set of all 2 to 1 bit functions, while the CLi 2 to 2 bit function is selected out of a restricted set. That means one programs a CAL in terms of boolean functions, while CLi is programmed in terms of selecting the connections between the available gates.
  * A full development system is more expensive: \$6,500–\$17,000 for PC vs. \$3,000 for PC(or \$10,000 for Sun).
  * The host MS–DOS host does not support standard large CA systems. Also, using the system through the net is more difficult. However, the company informed us that a Sun version of the development system will be available by the end of next year.
  * There is no possibility of programming CLi by using standard programming languages, but only through VLSI design languages (like ABEL, PLASM, state-machine-language, Verilog, Synopsys, VHDL) or the CAD system.
  * CLi is not seen as internal memory by the host, hence the access flexibility is limited. In contrast, CAL product comes with library routines allowing on-line access to internal chip information and with a debugging tool exploiting this facility.
  * There are several academic research groups using the CAL1024 chip for various purposes: systolic arithmetic, silicon compilation, computers with reconfigurable hardware. They believe that CAL offers a "cleaner" environment. There are no such users for CLi6000 (there will be 3–4 such users in the near future, acc. to a company statement).

- **Conclusion:** Suitable for or purpose.

9.3 Conclusions

The parallel hardware currently available within the ACPC network already offers some interesting alternatives for testing and implementing fine grain systolic algorithms for long integer/rational arithmetic, both on MIMD and SIMD architectures. The tests will be useful for the algorithms verification, they will bring more insight to the practical aspects of implementation, and will probably lead to a speed–up of the algebraic computations by a factor of 5 to 10.

However, the full efficiency of the systolic approach can be achieved only with more flexible computing structures, like the Field Programmable Gate Arrays (FPGA). After investigating several architectures with systolic structure, we come to the conclusion that the FPGA approach offers the best environment for testing and implementing fine grain systolic algorithms. It offers the flexibility of customized hardware, but without the disadvantages of the actual chip manufacturing process. This kind of hardware supports the implementation of the "active memory" computation paradigm, in which the memory itself is used as a computation agent. This new paradigm is likely to increase dramatically the hardware usage and to decrease the ratio price/performance for intensively computation tasks.

9.4 Sources of information on the hardware

- Sequent Symmetry:
    1. Product documentation and release notes.
- Impuls Multi–Transputer(T800):
    1. Product documentation and release notes.
    2. K. Kusche, W. Schreiner — *PC Guide*, RISC–Linz, Parallel Computation Laboratory, 1990.
- MasPar MP–1:
    1. *MP–1 family data–parallel computers*, MasPar Corp., 1990.
    2. *MasPar 1200 series computer systems*, MasPar Corp., 1991.
    3. *MasPar data–parallel programming languages*, MasPar Corp., 1990.
    4. T. Blank — *The MasPar MP–1 architecture*, MasPar Corp., 1990.
    5. P. Christy — *Software to support massively parallel computing on the MasPar MP–1*, MasPar Corp., 1990.
    6. G. Chinn, K. A. Grajski, C. Chen, C. Kuszmaul, S. Tomboulian — *Systolic array implementations of neural nets on the MasPar MP–1 massively parallel processor*, MasPar Corp., 1990.
- iWarp:

1. Kris Herbst — *Intel iWarp graduates from DARPA, offering up to 20 GFLOPS*, `supernet` article, April 1992.
2. *iWarp system configurations and options summary*, Intel Corp., 1991.
3. *Introduction to iWarp*, Intel Corp., 1991.
4. *Background information on iWarp real-time supercomputing for advanced signal and image processing*, Intel Corp., 1991.
5. H. T. Kung — *Computational models for parallel computers*, Phil. Trans. R. Soc. Lond. A 326, 357–371 (1988).
6. S. Bokar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, J. Webb — *Supporting systolic and memory communication in iWarp*, Proc. 17^{th} Ann. Int. Symp. on Comp. Sci., Seattle, 1990, IEEE Comp. Soc., 1990.
7. B. Greer — *A tutorial on using iWarp PathLib*, iWarp Tech. Note TN–1, Intel Corp., 1991.
8. H. T. Kung, J. Subhlok — *A new approach for automatic parallelization of blocked linear algebra computations*, Proc. Supercomputing 91, Albuquerque, Nov 1991.
9. D. R. O'Hallaron — *The* ASSIGN *parallel program generator*, Carnegie–Mellon Univ. Report CMU-CS-91-141, 1991.
10. L. G. C. Hamey, J. A. Webb, I-Chen Wu — *APPLY, a programming language for low-level vision on diverse parallel architectures*, in J. Kowalik (ed.) — *Parallel computation and computers for artificial intelligence*, Kluwer Academic Publishers, 1987.

- CAM–PC:

1. *Automatrix, Inc. introduces world's first single–board, 24 MIPS cellular automata machine for under $2000*, news release, Oct 1991.
2. *Product description sheet* and *Price sheet*, Algotronix, Inc., Feb 1992.
3. R. Tatar (CAM-PC Support Staff, by e-mail) — Various information on CAM–PC.

- CAL1024:

1. *CAL1024, CHS2x4 Custom Computer, Configurable Logic Software*, product briefs from Algotronix, Ltd., 1991.
2. C. Carruthers (Algotronix Ltd.) — *Configurable Array Logic (CAL) technology*, short product description and survey of customers.
3. C. Carruthers, private communications.
4. D. Pountain — *Algotronix: the first custom computer*, BYTE, Sept 1991.
5. W. Luk, I. Page — *Parametrising designs for FPGA*, in W. R. Moore, W. Luk (eds.) — *FPGA's*, Abbingdon EE&CS Books, 1991.
6. B. Heeb, C. Pfeister — *Chameleon: A workstation of a different color*, submitted to 2^{nd} Workshop on FPGA's, Aug 1992, Vienna.
7. C. Pfeister (by e-mail) — private communication.

- CLi6000:

1. *CLi6000 series Field-Programmable Gate Arrays*, Concurrent Logic, Inc., Apr 1992.
2. *Integrated Development System*, Concurrent Logic, Inc., Apr 1992.
3. *Integrated Development System Demonstration Disk*, Concurrent Logic, Inc., Apr 1992.
4. *Preis liste, Concurrent Logic*, Frontend Electronic Vertriebs GmbH, Apr 1992.
5. F. Furtek — *An FPGA architecture for massively parallel computing*, Concurrent Logic, Inc., 1992.

Chapter 10

Systolic multiplication on MasPar

We investigate the possibility of speeding-up multiplication by systolic implementation on a MasPar computer (SIMD architecture). For multiplication of multiprecision integers we obtain almost linear speed-up over the classical sequential algorithm (29 times for 30-word integers, efficiency 96%), using a serial–parallel systolic scheme on a linear array of processors. By embedding a variant of this algorithm in a systolic two-dimensional scheme for multiplication of univariate integral polynomials, we obtain almost half-linear speed-up (383 times for polynomials of degree 29 with multiprecision coefficients of 15 words, efficiency 43%). These experiments show the usefulness of the systolic paradigm for increasing the performance of algebraic computations.

Also, this research constitutes a relevant example of fruitful cooperation between two ACPC member organizations situated at different locations, by remote access to computing equipment, exchange of documentation, and useful discussions.

10.1 Introduction

Systolic parallelization of long integer arithmetic in the "most-significant digits first" (MSF) pipelined manner was considered by [Trivedi and Ercegovac, 1977] and other authors (see [Schwartzlander, 1990a], chapter 3), using the signed-digit redundant representation.

Our approach is different: we use "least-significant digits first" (LSF) algorithms, because this allows pipelined aggregation of various multiprecision operations (e.g. addition/subtraction units, on-line multiplication [Atrubin, 1965], exact division [Jebelean, 1993b, Jebelean, 1993d], LSF greatest common divisor [Brent and Kung, 1985]. Also, our algorithms use standard representation of multiprecision integers in an arbitrary radix (typically a power of 2), which makes them suitable for implementation on multiprocessor architectures. We demonstrate in this paper the possibility of embedding this kind of systolic algorithms into higher level algebraic algorithms (e.g. multiprecision multiplication into polynomial multiplication).

SIMD parallelization of computer algebra algorithms did not receive much attention in the literature. [Weeks, 1989] obtained a 45 times speed–up of Gröbner Basis computations by parallelizing multiprecision algorithms on the SIMD like Convex vector processor, but most of the speed-up is due to some improvements in list processing operations and to the use of 64-bit arithmetic. In particular, univariate polynomial multiplication by parallelizing both the level of coefficient operations and binary-digit operations was considered by [Beardsworth, 1981] on the ICL DAP computer (SIMD architecture).

The algorithms we use are multiprecision variants of serial / parallel multipliers which can be easily derived from the "school method" for multiplication. Apparently these algorithms were the first to be considered for hardware multiplication – see [Booth, 1951]. In both algorithms, one of the operands is present in the array at the beginning of computation, while the elements of the other one are broadcasted to the parallel processors, one at each step. The first algorithm pipelines the result out via the first processor, while the second algorithm leaves the result in the array. The second algorithm is suitable for embedding into polynomial multiplication scheme, yielding an algorithm with two–level systolic parallelism, which maps naturally onto the two–dimensional architecture of MasPar.

The experimental results show the effectiveness of the systolic paradigm for increasing the efficiency of algebraic computations. For multiplication of multiprecision integers we obtain almost linear speed-up over the classical sequential algorithm (29 times for 30 digit integers, efficiency 95%), using a systolic parallelization scheme for a linear array of processors. The two-dimensional algorithm for multiplication of univariate integral polynomials gives almost half-linear speed-up (383 times for polynomials of degree 29 with multiprecision coefficients of 15 digits, efficiency 43%).

Last but not least, this research was done in the frame of a joint research project of the ACPC (Austrian Center for Parallel Computation), and constitutes a relevant example of cooperation between two member organizations situated at different locations. The author (from RISC–Linz) ran the experiments remotely on the MasPar computer of the Technical University of Vienna. This activity was facilitated by the organization of the ACPC network, which allows researchers from the member organizations to use all the computing facilities of the other members, and by the constant exchange of documentation and informations between the ACPC members. Also, the author benefited from useful discussions and help from the other MasPar users, occasioned by the regular ACPC workshops and meetings. This shows that organization of cooperation frames between academic groups with similar interests (in our case: parallel computing) and starting joint projects involving several such groups is likely to add new potential to the research activity.

10.2 Sequential classical multiplication

10.2.1 Integers

The input to the multiplication algorithm consists of two multiprecision integers A, B represented as lists of positive digits in radix $\beta = 2^{29}$, least-significant digits first. The output is the multiprecision integer $C = A * B$.

Suppose:

$$A = a_0 + a_1 * \beta + \ldots + a_{n-1} * \beta^{n-1},$$

$$B = b_0 + b_1 * \beta + \ldots + b_{m-1} * \beta^{m-1},$$

$$C = c_0 + c_1 * \beta + \ldots + c_{n+m-1} * \beta^{n+m-1}.$$

Then

$$C = (a_0 + a_1 * \beta + \ldots + a_{n-1} * \beta^{n-1}) * B,$$

that is:

$$C = (a_0 * B) + (a_1 * B) * \beta + \ldots + (a_{n-1} * B) * \beta^{n-1}.$$

```
[0] C ← IntClasMul(A, B) [Classical integer multiplication]
[1]     C ← (0, ..., 0) [m + n positions]
[2]     for i = 0, 1, ..., n − 1 do
[3]         carry ← 0
[4]         for j = 0, 1, ..., m − 1 do
[5]             (carry, c_{i+j}) ← c_{i+j} + b_j * a_i + carry
[6]         if carry ≠ 0
[7]             then c_{i+m} ← c_{i+m} + carry
```

Figure 10.1: Classical multiprecision multiplication

Hence, one can initialize $C = (0, \ldots, 0)$, and then, in n steps ($0 \leq i \leq n-1$), one can compute the lists $a_i * B$ and add them to C, starting at the position of c_i. At each individual step a $carry$ is produced, which must be added to the next digit of C.

The description of the algorithm is presented in Fig.10.1.

Note that the addition in line [7] is not really necessary here. Indeed, at the end of the i^{th} step, C contains the product $(a_0+a_1*\beta+\ldots+a_{i-1}*\beta^{i-1})*B$, which is the product of an i digits integer by an m digits integer. This product has at most $i+m$ nonzero digits, hence c_{i+m} is zero. However, this addition will be necessary when we embed this algorithm into polynomial multiplication, for computing $C + A * B$.

The theoretical time complexity of this algorithm depends on the innermost loop (line [5]), which is performed $n * m$ times, hence

$$T_{\text{classic}} = O(n * m).$$

For balanced–length operands

$$T_{\text{classic}} = O(n^2).$$

10.2.2 Polynomials

The input to the multiplication algorithm consists of two integral univariate polynomials $\mathcal{A}, \mathcal{B}$, represented as lists of multiprecision integers, least-degree coefficients first. The output is the polynomial $\mathcal{C} = \mathcal{A} * \mathcal{B}$.

Let us denote

$$\mathcal{A} = A_0 + A_1 * x + \ldots + A_{N-1} * x^{N-1},$$

[0] $\mathcal{C} \leftarrow$ `PolyClasMul`$(\mathcal{A}, \mathcal{B})$ [Classical polynomial multiplication]
[1] $\quad \mathcal{C} \leftarrow (0, \ldots, 0)$ [$M + N - 1$ positions]
[2] $\quad$ `for` $I = 0, 1, \ldots, N - 1$ `do`
[3] $\quad\quad$ `for` $J = 0, 1, \ldots, M - 1$ `do`
[4] $\quad\quad\quad C_{I+J} \leftarrow C_{I+J} + B_J * A_I$

Figure 10.2: Classical multiplication of polynomials

$$\mathcal{B} = B_0 + B_1 * x + \ldots + B_{M-1} * x^{M-1},$$
$$\mathcal{C} = C_0 + C_1 * x + \ldots + C_{N+M-2} * x^{N+M-2}.$$

Then
$$\mathcal{C} = (A_0 + A_1 * x + \ldots + A_{N-1} * x^{N-1}) * \mathcal{B},$$
that is
$$\mathcal{C} = (A_0 * \mathcal{B}) + (A_1 * \mathcal{B}) * x + \ldots + (A_{N-1} * \mathcal{B}) * x^{N-1}.$$

Similarly to the case of integers, one obtains the algorithm presented in Fig.10.2.

The innermost loop of this algorithm performs the operation $C + A * B$ over long integers, which is exactly what `IntClasMul` does, when the initialization in line [1] is removed.

The innermost loop (line [4]) is performed $N * M$ times, and it requires $O(n * m)$ steps, where n, m are the maximum lengths of the coefficients of $\mathcal{A}, \mathcal{B}$. Hence
$$T_{\text{classic}} = O(N * M * n * m).$$
For balanced–length operands
$$T_{\text{classic}} = O(N^2 * n^2).$$

10.3 Systolic algorithms

10.3.1 The MasPar computer

In designing the parallel algorithm, we make use of the specific architecture of MasPar. We present here only those features which are relevant for our approach. More details can be found in [MasPar90, 1990, MasPar92, 1992]

```
int a;
plural int b, c;
...
a = a*5;
if(b==0)
   c = c/2;
```

Figure 10.3: A sample MasPar program

MasPar is a SIMD distributed memory machine, with 1024 Processing Elements (PE's), arranged in a 32 by 32 mesh (torus). The PE's are driven by a sequential Array Control Unit (ACU). The device can be programmed in the language C, with some special extensions for handling MasPar parallelism:

- Data: The ACU and each PE have their own internal memory for data. In C language, one has to use `plural` to declare the variables which are allocated on the parallel PE's. A `plural` variable will have one instance on each PE, possibly containing different values. The variables which are not `plural` are called *singular* and are allocated on ACU.

- Program flow: The operations involving only *singular* variables are executed sequentially on the ACU. The operations involving `plural` variables are executed in parallel on the PE's. All PE's execute the same instructions synchronously. However, at certain moments some of the PE's may be "masked", and then they execute nothing. The "masking" is done by using C language conditional constructs. For instance, if the program is like in Fig.10.3, then `a` is a *singular* variable, and `b, c` are `plural` variables. Hence, "`a = a*5;`" is executed on ACU, and then "`c = c/2;`" is executed on those PE's on which `b` is zero (they are called *active* PE's). The other PE's are "masked" and execute nothing at this step.

- Data communication between ACU and PE's: The ACU accesses all the PE's in parallel. Hence, data can be broadcasted to all the PE's in one step. For instance, if `a` is *singular* and `b` is `plural`, then the assignment "`b = a;`" will send the value of `a` to all active PE's. The reverse operation is more delicate, because `b` might have different values on different processors. One may use "`a = globalor(b);`", which performs a bitwise logical OR on all `b's` in the active PE's. Also, a plural variable (say `b`) on a particular PE can be accessed using `proc[i].b` (linear addressing, $0 \leq$ `i` ≤ 1023) or `proc[y][x].b` (2D addressing, $0 \leq \mathtt{x}, \mathtt{y} \leq 31$). The `proc` construct may be used in either left or right hand side of assignments, thus yielding the means to store/load values to/from particular PE's.

- Data communication between PE's: Data may be moved between adjacent PE's by using the `xnet` construct. For instance, if `b` and `c` are plural variables, then "`xnetW[1].b = c;`" means "store the value of `c` into `b` of the left neighbor". This is also executed in parallel on all active PE's. A similar result can be obtained with "`b = xnetE[1].c;`". Any of the 8 directions can be used (horizontally, vertically, diagonally). At the PE array borders the data wraps around in a torus like fashion.

10.3.2 Integers

Let us modify the algorithm `IntClasMul` such that the inner loop can be executed in parallel. In the present form of the algorithm, the inner loop is inherently sequential, because each step uses the *carry* produced by the previous step. Therefore, we shall devise a different scheme for *carry* propagation. We use a list $Y = (y_0, y_1, \ldots, y_{n+m-1})$ to hold the carries produced at each step. This list has as many elements as C has. The computation then proceeds in two stages:

- Stage 1: The additions and the multiplications are performed and the carries are produced. Since the outermost loop (over A) is performed sequentially, the carries produced in one step may be used in the following step.

- Stage 2: The carries are absorbed into C, by adding each y_k to c_{k+1}. These additions may produce new carries, which are again stored in Y list, and the absorption stage is repeated until all the carries become zero.

The parallel algorithm is presented in Fig.10.4

[0] $C \leftarrow$ `IntParMul`(A, B) [Parallel integer multiplication]
[1] $\quad C, Y \leftarrow (0, \ldots, 0)$ [$m+n$ positions]
[2] $\quad$ **for** $i = 0, 1, \ldots, n-1$ **do** [Stage 1: add and multiply]
[3] $\quad\quad$ **for** $j = 0, 1, \ldots, m-1$ **in parallel do**
[4] $\quad\quad\quad (y_{i+j+1}, c_{i+j}) \leftarrow c_{i+j} + b_j * a_i + y_{i+j}$
[5] $\quad$ `while(`*there are nonzero carries*`)` **do** [Stage 2: absorb carries]
[6] $\quad\quad$ **for** $j = n, n+1, \ldots, n+m-1$ **in parallel do**
[7] $\quad\quad\quad (y_{j+1}, c_j) \leftarrow c_j + y_j$
[8] $\quad\quad y_n \leftarrow 0$

Figure 10.4: Parallel integer multiplication

One can see that m processors are necessary to implement this algorithm. During Stage 1, the m processors act as a window which moves along the vector C, one element at a time. In other words, C is piped through the string of m processors. During Stage 2, the window is fixed on the last m elements of C.

The first version of the systolic algorithm (Fig.10.5) is a slight modification of this parallel algorithm. In order to make SIMD implementation easier, we emphasize the local variables on each processor. These variables are denoted by $\overline{B} = (\overline{b}_0, \overline{b}_1, \ldots, \overline{b}_m)$, $\overline{C} = (\overline{c}_0, \overline{c}_1, \ldots, \overline{c}_m)$, $\overline{Y} = (\overline{y}_0, \overline{y}_1, \ldots, \overline{y}_m)$. The vector A is not stored in the processors. Rather, at each iteration of the main loop, one element of A is send to all the processors for the computation in line [6]. We also introduce an $m + 1^{th}$ processor whose $\overline{b}_m, \overline{c}_m$, and $\overline{y}_m$ are zero all the time. This boundary processor does not participate in the computation, but its presence avoids boundary tests.

Let us find the time complexity of this algorithm. The parallel loops [5], [8], [11] and the initialization [1] require constant time. The other loops are [2]: n steps, [4]: n steps, [10]: at most m steps, and [14]: m steps. Hence

$$T_{\text{systolic}} = O(n+m).$$

For balanced–length operands:

$$T_{\text{systolic}} = O(n).$$

In fact, for large β, the `while` loop [10] usually requires only 2 steps. Indeed, after the first step, the carries can be at most 1. The probability that more

$C \leftarrow$ `IntSysMul.1`(A, B) [Systolic integer multiplication, version 1]
    $\overline{B}, \overline{C}, \overline{Y} \leftarrow (0, \ldots, 0)$ [$m+1$ positions]
    `for` $j = 0, 1, \ldots, m-1$ `do` [load B sequentially]
        $\overline{b}_j \leftarrow b_j$
    `for` $i = 0, 1, \ldots, n-1$ `do` [Stage 1: add and multiply]
        `for` $j = 0, 1, \ldots, m-1$ `in parallel do`
            $(\overline{y}_j, \overline{c}_j) \leftarrow \overline{c}_j + \overline{b}_j * a_i + \overline{y}_j$ [compute]
            $c_i \leftarrow \overline{c}_0$ [extract next digit of C]
        `for` $j = 0, 1, \ldots, m-1$ `in parallel do`
            $\overline{c}_j \leftarrow \overline{c}_{j+1}$ [shift $\overline{C}$ left]
    `while globalor(`$\overline{Y}$`) do` [Stage 2: absorb carries]
        `for` $j = 0, 1, \ldots, m-1$ `in parallel do`
            $(\overline{y}_{j+1}, \overline{c}_j) \leftarrow \overline{c}_j + \overline{y}_j$
        $\overline{y}_0 \leftarrow 0$
    `for` $j = 0, 1, \ldots, m-1$ `do` [extract rest of C sequentially]
        $c_{n+j} \leftarrow \overline{c}_j$

Figure 10.5: Systolic multiprecision multiplication, version 1.

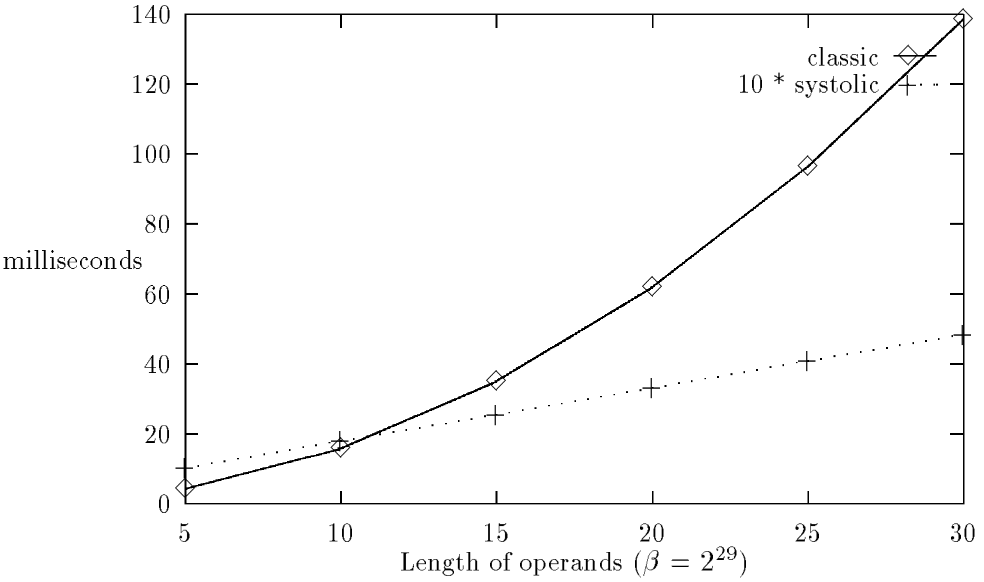

Figure 10.6: Comparative timings for multiprecision multiplication.

than 1 step is necessary to absorb them is less than the probability that at least one of the elements of C is $\beta - 1$, which probability is lower than m/β.

We implemented the algorithm `IntSysMul.1` on MasPar MP1, using only the first row of 32 processors. In order to evaluate the speed–up, we also implemented the algorithm `IntClasMul`, using only one processor.

Fig.10.6 shows the timings in milliseconds for the two algorithms (systolic timings are scaled by 10). The speed–up (systolic time over classical time), and the efficiency (speed–up over number of processors) are shown in Fig. 10.7 and Fig. 10.8 . The efficiency was computed taking into account only those processors which actually participate in the computation (that is m).

If C is to be used in subsequent computations, then it is useful to leave it in the array, instead of pipelining/extracting it. Using $n+m+1$ processors, C can be stored in $\overline{C}$, and then $\overline{B}$ and $\overline{Y}$ must be shifted rightward one position at each step. This second version of the systolic algorithm is presented in Fig.10.9.

The theoretical complexity is the same as before. However, in steps [5]–[7] only m of the $n + m + 1$ processors do useful work. This results in a

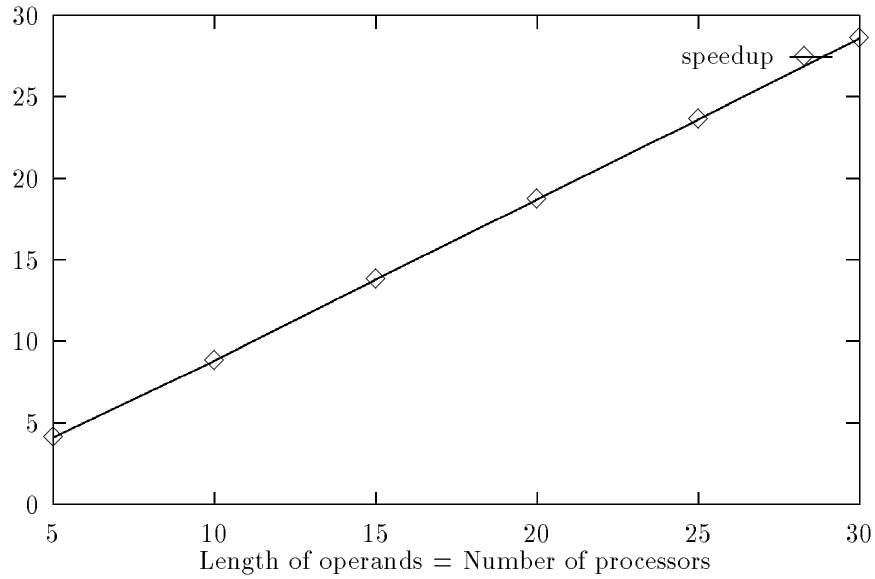

Figure 10.7: Speed–up of systolic multiprecision multiplication.

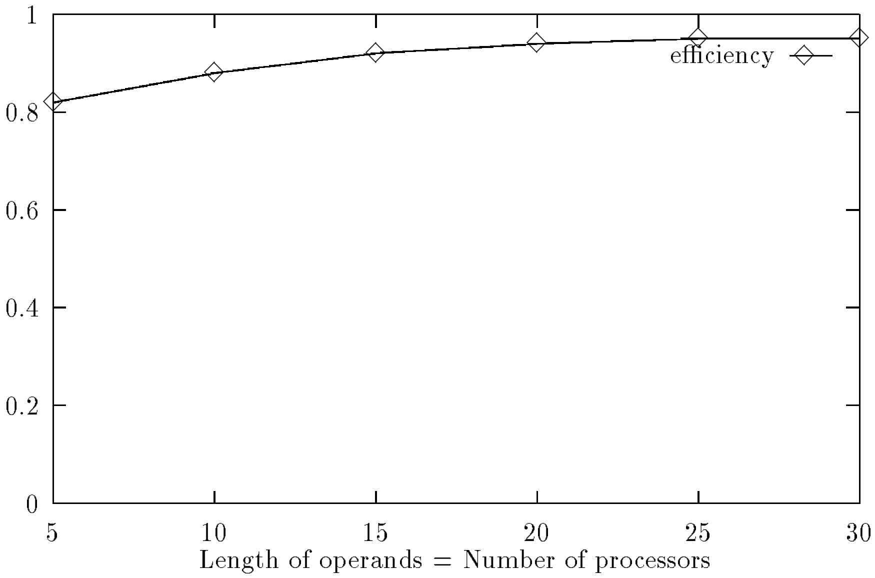

Figure 10.8: Efficiency of systolic multiprecision multiplication.

[0] $C \leftarrow$ `IntSysMul.2`(A, B) [Systolic integer multiplication, version 2]
[1] $\quad \overline{B}, \overline{C}, \overline{Y} \leftarrow (0, \ldots, 0)$ [$n + m + 1$ positions]
[2] $\quad$ `for` $j = 0, 1, \ldots, m - 1$ `do` [load B sequentially]
[3] $\quad\quad \overline{b}_j \leftarrow b_j$
[4] $\quad$ `for` $i = 0, 1, \ldots, n - 1$ `do` [Stage 1: add and multiply]
[5] $\quad\quad$ `for` $j = 0, 1, \ldots, n + m - 1$ `in parallel do`
[6] $\quad\quad\quad (\overline{y}_{j+1}, \overline{c}_j) \leftarrow \overline{c}_j + \overline{b}_j * a_i + \overline{y}_j$ [compute, shift $\overline{Y}$]
[7] $\quad\quad\quad \overline{b}_{j+1} \leftarrow \overline{b}_j$ [shift $\overline{B}$]
[8] $\quad$ `while globalor(`$\overline{Y}$`) do` [Stage 2: absorb carries]
[9] $\quad\quad$ `for` $j = 0, 1, \ldots, n + m - 1$ `in parallel do`
[10] $\quad\quad\quad (\overline{y}_{j+1}, \overline{c}_j) \leftarrow \overline{c}_j + \overline{y}_j$
[11] $\quad$ `for` $j = 0, 1, \ldots, n + m - 1$ `do` [extract C sequentially]
[12] $\quad\quad c_j \leftarrow \overline{c}_j$

Figure 10.9: Systolic multiprecision multiplication, version 2.

lower efficiency of parallelism (1/2 for balanced-length operands). On the other hand, having C steady in the processors allows simple embedding of this algorithm into systolic algorithms for operation with polynomials.

10.3.3 Polynomials

In this case there is no problem in parallelizing the inner loop of the algorithm `PolyClasMul` (fig. 10.2), using M computing units. As in the case of `IntSysMul.1` (fig. 10.5), $\mathcal{C}$ is piped through the string of these M computing units.

However, each of these computing units must be able to compute $C+B*A$ on long integers. This is exactly what is done by the algorithm `IntSysMul.2`, if the initialization of C is removed from line [1] (Fig.10.9). Therefore, we shall use a row of processors for each of the above M computing units. Overall, we need a matrix of $(M + 1) * (n + m + 2)$ processors, where n, m are the maximum lengths of the coefficients of $\mathcal{A}, \mathcal{B}$.

The local variables of each processor are denoted by $\overline{\mathcal{B}} = (\overline{b}_{J,j})$, $\overline{\mathcal{B}}' = (\overline{b'}_{J,j})$, $\overline{\mathcal{C}} = (\overline{c}_{J,j})$, $\overline{\mathcal{Y}} = (\overline{y}_{J,j})$. $\overline{\mathcal{B}}'$ contains the coefficients of $\mathcal{B}$, shifted rightward as required by the algorithm `IntSysMul.2`, and gets from $\overline{\mathcal{B}}$ the

[0] $\mathcal{C} \leftarrow$ `PolySysMul`$(\mathcal{A}, \mathcal{B})$ [Systolic polynomial multiplication]
[1] $\quad \overline{\mathcal{C}}, \overline{\mathcal{B}}, \overline{\mathcal{Y}} \leftarrow (0, \ldots, 0)$ [in parallel]
[2] $\quad$ `for` $(J, j) = (0,0), (0,1), \ldots, (M-1, m-1)$ `do` [load $\mathcal{B}$ sequentially]
[3] $\qquad \overline{b}_{J,j} \leftarrow b_{J,j}$
[4] $\quad$ `for` $I = 0, 1, \ldots, N-1$ `do` [scan coefficients of $\mathcal{A}$]
[5] $\qquad$ `for` $J = 0, 1, \ldots, M-1$ `in parallel do` $[C_{I+J} \leftarrow C_{I+J} + B_J * A_I]$
[6] $\qquad\quad \overline{b}'_{J,j} \leftarrow \overline{b}_{J,j}$ [restore $\overline{\mathcal{B}}$]
[7] $\qquad\quad$ `for` $i = 0, 1, \ldots, n-1$ `do` [Stage 1: add and multiply]
[8] $\qquad\qquad$ `for` $j = 0, 1, \ldots, n+m$ `in parallel do`
[9] $\qquad\qquad\quad (\overline{y}_{J,j+1}, \overline{c}_{J,j}) \leftarrow \overline{c}_{J,j} + \overline{b'}_{J,j} * a_{I,i} + \overline{y}_{J,j}$ [compute, shift $\overline{\mathcal{Y}}$]
[10] $\qquad\qquad\quad \overline{b'}_{J,j+1} \leftarrow \overline{b'}_{J,j}$ [shift $\overline{\mathcal{B}}$]
[11] $\qquad\quad$ `while globalor(`$\overline{\mathcal{Y}}$`) do` [Stage 2: absorb carries]
[12] $\qquad\qquad$ `for` $j = 0, 1, \ldots, n+m$ `in parallel do`
[13] $\qquad\qquad\quad (\overline{y}_{J,j+1}, \overline{c}_{J,j}) \leftarrow \overline{c}_{J,j} + \overline{y}_{J,j}$
[14] $\qquad$ `for` $j = 0, 1, \ldots, n+m$ `do` [extract C_I sequentially]
[15] $\qquad\quad c_{I,j} \leftarrow \overline{c}_{0,j}$
[16] $\qquad$ `for` $(J, j) = (0,0), (0,1), \ldots, (M-1, m-1)$ `in parallel do`
[17] $\qquad\quad \overline{c}_{J,j} \leftarrow \overline{c}_{J+1,j}$ [shift $\overline{\mathcal{C}}$ upwards]
[16] $\quad$ `for` $J = 0, 1, \ldots, M-1$ `do` [extract rest of $\mathcal{C}$ sequentially]
[17] $\qquad$ `for` $j = 0, 1, \ldots, n+m$ `do` [extract C_{N+J} sequentially]
[18] $\qquad\quad c_{N+J,j} \leftarrow \overline{c}_{J,j}$

Figure 10.10: Systolic polynomial multiplication.

non–shifted values at the beginning of each main cycle. The $(M+1)^{th}$ row and the $(n+m+2)^{th}$ column of processors ensure the boundary conditions and do not participate in the computation. Their local variables equal zero all the time. The coefficients of $\mathcal{A}$ are not stored in the parallel processors. Rather, they are send to all the processors, one digit at each step, for the computation in line [9] (see Fig.10.10).

Let us compute the time complexity. We do not count the parallel loops [5], [8], [12] and [16]. The loop [2] is performed $M * m$ times, the loop [4] (N times) has several inner loops: [7] n times, [11] at most m times, and [14] $n + m$ times, hence $N * (n+m)$ is dominating. Finally, the loop [16] is

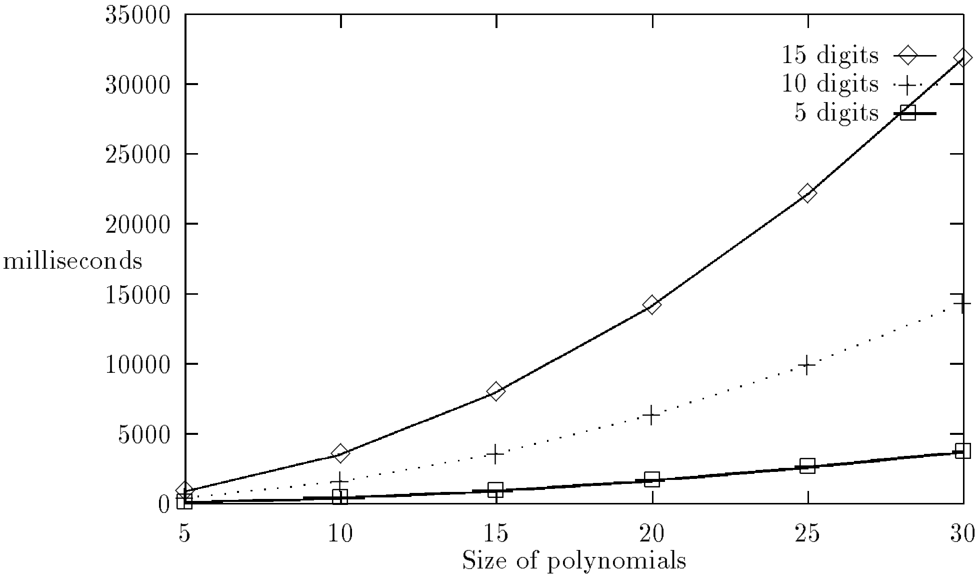

Figure 10.11: Timings of classical polynomial multiplication.

repeated $M * (n + m)$ times. All in all:

$$T_{\text{systolic}} = O((N + M) * (n + m)).$$

For balanced–length operands:

$$T_{\text{systolic}} = O(N * n).$$

We implemented the algorithm `PolySysMul` on MasPar MP1, using the matrix of 32 by 32 processors. In order to evaluate the speed–up, we also implemented the algorithm `IntClasMul`, using only one processor.

Figures 10.11, 10.12, 10.13 and 10.14 show the timings, the speed–up, and the efficiency. The efficiency was computed taking into account only those processors which actually participate in the computation (that is $M * (n + m)$).

Conclusions

The experimental curves match well the theoretical expectations:

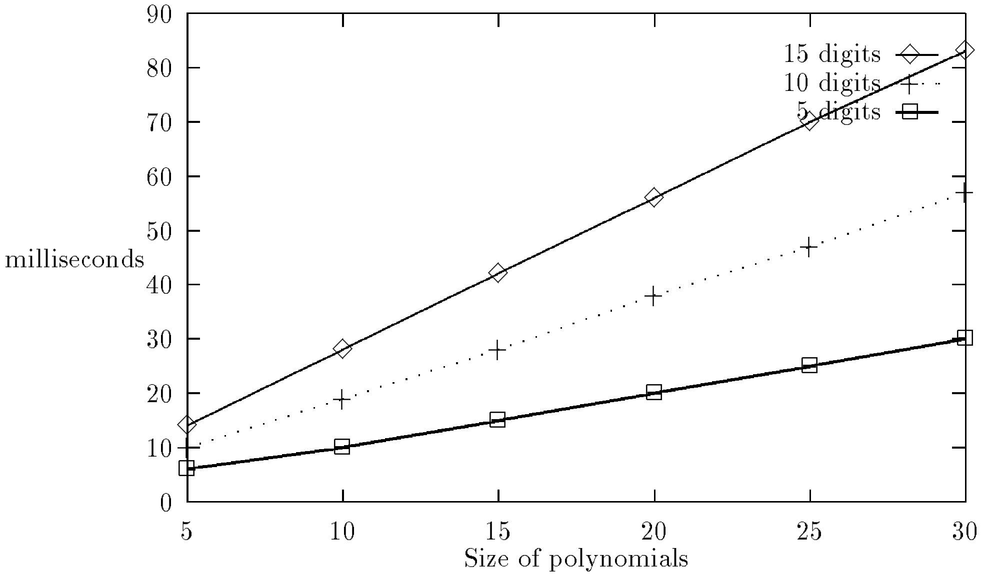

Figure 10.12: Timings of systolic polynomial multiplication.

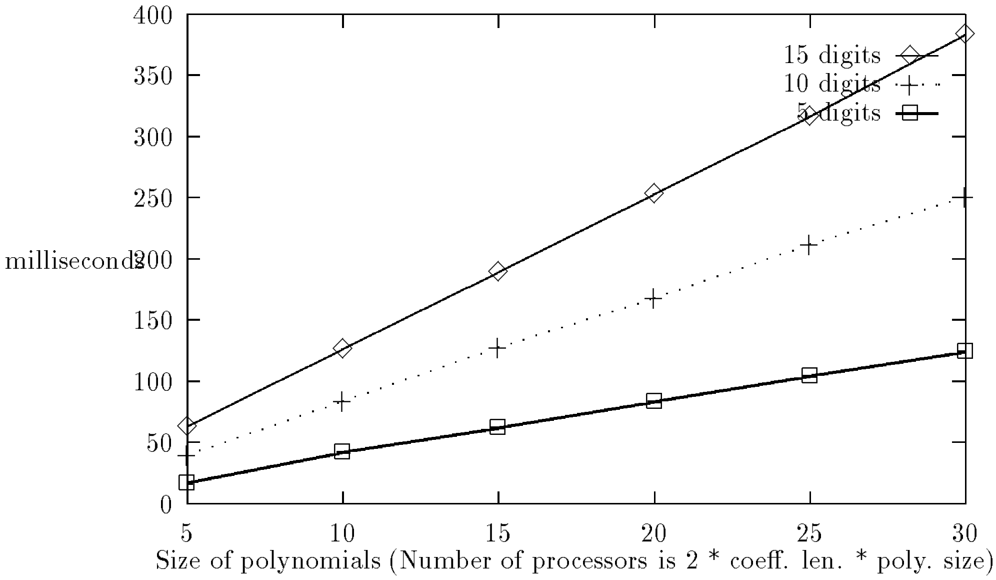

Figure 10.13: Speed-up of systolic polynomial multiplication.

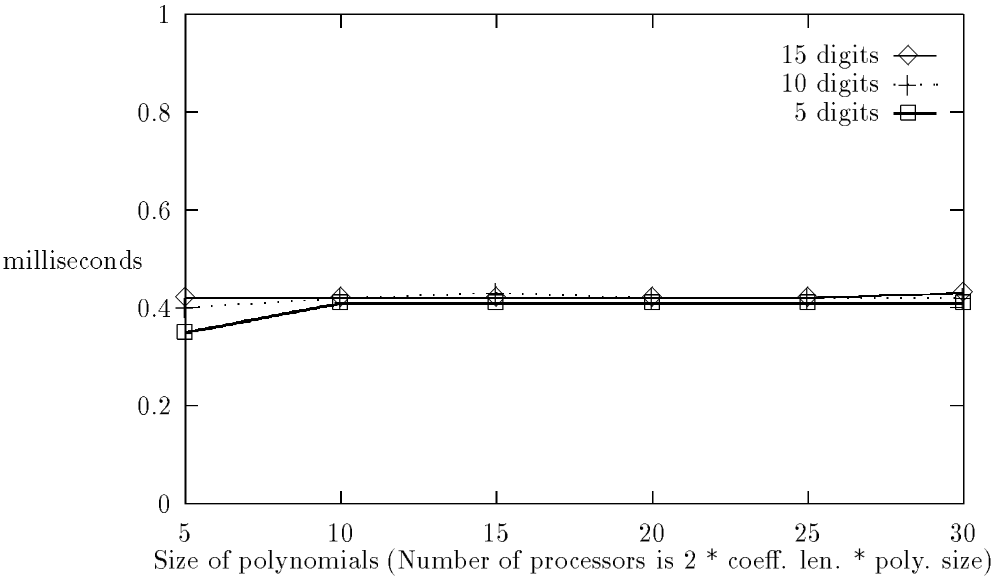

Figure 10.14: Efficiency of systolic polynomial multiplication.

- quadratic time for classic algorithm;
- linear time for systolic algorithm;
- linear speed–up;
- constant efficiency.

The high efficiency obtained for long integer multiplication demonstrates the effectiveness of the systolic parallelization approach. Even in the case of polynomial multiplication, the efficiency approaches the 50% theoretical limit, due to the design of `IntSysMul.2`. We believe this efficiency is also quite good in this case, considering the high number of processors used and the inherent limitations of the SIMD model. In fact, the performance is not so bad compared with the 37% to 86% performance shown by the most popular parallel computers with more than 500 processors on the *LINPACK* benchmark, which is a quite regular problem – see [Dongarra, 1993].

When considering the problem from a strictly practical point of view, one should interpret this results with caution. Namely, measuring the speed–up against the running time of the classic algorithm on one PE gives insight about the efficiency of the algorithm, but not about the practical speed–up which can be obtained for a real-life application. We timed the classic algorithm on a fast sequential machine (DECstation 5200), using a good optimizing compiler (GNU C), and for practically the same code we obtained a speed–up of 80 to 100 over the MasPar PE runs. This means that only a fraction of the speed–up obtained by running on the MasPar are still valid from the pure practical point of view.

Chapter 11

Rational arithmetic on MasPar

We investigate the possibility of systolic parallelization of the arithmetic of multiprecision rational numbers using the MasPar computer.

We implement multiprecision rational arithmetic using the known systolic algorithms for long integers addition and multiplication, as well as new systolic algorithms for exact division and GCD computation. Both new algorithms have the advantage that they work in "least-significant digits first" pipelined manner, hence they can be well aggregated with the systolic algorithms for addition and multiplication.

The practical experiments show that the timings are linearly dependent of the lengths of the inputs, hence demonstrating the effectiveness of the systolic paradigm for implementing long integer arithmetic on SIMD architectures.

11.1 Introduction

This chapter describes a systolic implementation of operations with long rationals on the MasPar computer (SIMD architecture). The features of MasPar which are interesting for our application were already presented in section 10.3.1.

The operation which we implement is *rational reduction*, that is, given rational numbers

$$\frac{A}{B}, \ \frac{C}{D}, \ \frac{X}{Y},$$

find

$$\frac{E}{F} = \frac{A}{B} - \frac{X}{Y} * \frac{C}{D},$$

where E/F is normalized. This operation is heavily used, for instance, in Gröbner bases computation, for inter-reduction of polynomials.

All the basic operations with long integers are involved in this reduction:

$$E' = A * Y * D - B * X * C,$$

$$F' = B * Y * D,$$

$$G = GCD(E', F'),$$

$$E = E'/G, F = F'/G,$$

where the last two divisions are *exact divisions*.

The computations are performed using a *linear* array of processors, each processor containing one digit of each operand. One *digit* is a 32-bit word.

The input/output is intermingled with the computations as follows:

- the operands loaded into the array during multiplication;
- GCD is computed between operands present in the array, and gives the result also in the array;
- exact division is performed between operands present in the array, but outputs the result during computation.

This organization demonstrates the possibility of overlapping computation with I/O operations, which is one of the advantages specific to systolic algorithms.

The long integers involved in the computations are represented as arrays of *positive* digits (32-bit words). The *signs* of the integers are handled separately. *Complement* representation is temporary used when a subtraction has to be performed.

11.2 Multiplication and addition

Each **multiplication** is performed according to the scheme presented in figure 10.9 (section 10.4). These scheme requires one operand to be present in the array, while the other is loaded during multiplication. Therefore, first C and D are sequentially loaded into the array (this is the only I/O operation which does not overlap with actual computation). Subsequently, $Y * D$ and $X * C$ are computed (first stage), then $A * Y * D$, $B * Y * D$ and $B * X * C$ (second stage). Note that some communications can be spared because of the common operands occurring in the second multiplication stage.

Addition (subtraction) is performed using complement representation. Namely, the digits of the inputs to (and outputs from) addition are *positive*, no matter which are the actual signs of the operands. Before addition, the signs are examined and the second operand is complemented if an actual subtraction has to be performed. The operands are represented by filling-up the array with additional words (*sign-words*) which equal zero for a positive operand and $2^{32} - 1$ for a negative operand. After that, a simple addition will perform the desired operation. If the result is negative (this can be determined by inspecting the sign-words), than it is complemented again, and the information about its sign is updated accordingly.

The actual addition is performed in digit-parallel fashion, using the ripple-carry scheme. This scheme has a linear-time worst-case complexity, but, however, the probability of the worst-case situation is extremely low when using high-radix digits. In fact, in our experiments we never encountered a situation when the carry propagation needed more than 2 steps.

However, there is a situation when the carry systematically ripples along many words. This is the case of subtraction when the result is positive: then the carry ripples over the sign-words. In order to limit the number of steps in this case, we use a *mask*, which indicates the interesting words of the result. Namely, the value of *mask* is 1 in all the processors containing significant digits of the result, including the processor containing the least-significant sign-word, and 0 in the rest of the processors. During digit-parallel addition, the carry will get the value of the *mask* each time an overflow is detected.

The detailed algorithm is presented in Fig.11.1. The notations are those used in section 10.4. L_A, L_B, L_C tell lengths and signs of A, B, C. Y holds the carries, M is the mask. a_i, b_i, c_i are the digits of A, B, C stored in processor P_i. Likewise, y_i, m_i are the elements of Y, M stored in processor P_i.

Although the description of the algorithm seems more complicated that

```
(C, L_C) ← IntSysAdd(A, L_A, B, L_B) [C ← A + B]
    Y ← (0, ..., 0) [init carries]
    S_A ← sign(L_A) [sign of A]
    S_B ← sign(L_B) [sign of B]
    L_C ← 1 + max(abs(L_A), abs(L_B)) [length of C]
    for i = 0, 1, ... in parallel do [set mask]
        if i > L_C
            then m_i ← 0
            else m_i ← 1
    if S_A * S_B < 0 then[it is a subtraction: complement B]
        for i = 0, 1, ... in parallel do
            b_i ← not(b_i) [invert]
        y_0 ← 1 [add 1 to B]
        while globalor(Y) do [absorb carries in not(B) + 1]
            for i = 0, 1, ... in parallel do
                (y_{i+1}, b_i) ← b_i + y_i
    for i = 0, 1, ... in parallel do [actual addition ...]
        (y_{i+1}, c_i) ← a_i + b_i[...of A and B]
    while globalor(Y) do [absorb carries in C = A + B]
        for i = 0, 1, ... in parallel do
            (y_{i+1}, c_i) ← c_i + y_i
    i ← L_C [check real length of result]
    while i > 0 and c_i = c_{i-1} [look for least-significant sign-word]
        i ← i - 1
    if c_i ≠ 0 then [sign-word nonzero means result is negative]
        L_C ← -L_C [update sign]
        for i = 0, 1, ... in parallel do [complement result]
            c_i ← not(c_i) [invert]
        y_0 ← 1 [add 1 to C]
        while globalor(Y) do [absorb carries in not(C) + 1]
            for i = 0, 1, ... in parallel do
                (y_{i+1}, c_i) ← c_i + y_i
    L_C ← S_A * L_C [update sign of C]
```

Figure 11.1: Systolic multiprecision addition.

the one of multiplication, the operations involved (additions) are cheap in comparison with multiplications, and the experimentally measured running time is in fact constant. Within the entire rational reduction operation, addition takes less than 0.1% of the time.

11.3 Greatest Common Divisor

GCD computation is the most complicated and also the most time-consuming operation. Also, parallelization of the classical Euclidean algorithm (or Lehmer improved scheme) in systolic fashion (in fact in any fashion) is difficult, because of the carry propagation. In order to avoid the carry propagation problem one has to use an algorithm in which the decisions are taken using the *least-significant* digits of the operands. Such an algorithm is the *binary* algorithm of [Stein, 1967], which was adapted for systolic computations by [Brent and Kung, 1985] - the so called PlusMinus algorithm. The last algorithm, however, works at *binary* level, hence it is not suitable for implementation on multiprocessor machines working at *word* level.

Therefore, we parallelize here the *generalized binary* algorithm from chapter 6, which works least-significant digits first, and also at word level. As noted in the conclusions of chapter 6, the algorithm needs some further adaptations in order to be suitable for systolic parallelization. Namely, the problem is that the generalized binary algorithm finds an *approximation* G' of the true $G = GCD(A, B)$, which in the sequential version is corrected by computing

$$G = GCD(A, B, G') = GCD(A \bmod G', B \bmod G', G').$$

These computations (division with remainder, GCD by Lehmer-Euclid algorithm) are difficult to parallelize systolically, hence we want to avoid them. One way would be to replace the division with remainder by exact division, whose result is also suitable for finding the true GCD, and then continue the computation using the systolic PlusMinus algorithm or an improved version for high-radix computation.

We do not use this approach here, but an exact version of the generalized binary algorithm. The only cause for this is the simplicity of the implementation. The other variant might be more efficient, but this can only be determined by experiments.

As the reader remembers from section 6.3, the man idea of the generalized binary algorithm is to iterate a reduction step consisting in replacing the two operands A, B by $B, xA + yB$, where x, y are the modular conjugates of the least significant digits a, b of A, B. The cofactors x and y are obtained by applying the extended Euclidean algorithm to $2^{2w}, (a/b) \bmod 2^{2w}$, where w is the length of the word. As described in section 6.2 (and with those notations), x and y are v_k and a_k at the step k at which a_k becomes smaller then 2^w. This ensures that $x, |y| < 2^w$. The transformation is not GCD preserving, therefore an approximation of the GCD is obtained.

In order to obtain the true GCD, we retrieve *two pairs of cofactors* from the extended Euclidean algorithm, namely x, y and x', y'. The first pair of cofactors x, y is the same as in the original algorithm (v_k, a_k), while the second pair x', y' is just the previous one in the extended Euclidean algorithm (v_{k-1}, a_{k-1}). The fact that the transformations in the extended Euclidean algorithm are GCD preserving extends in a straight forward manner to the fact that the transformation

$$(A, B) \leftarrow (xA + yB, x'A + y'B).$$

is also GCD preserving.

However, the "smallness" condition does not hold anymore for the new cofactors $(x'y')$. Therefore, in order to avoid multiplication with 2-word cofactors, we use the value $w = 16$, that is, we compute cofactors for single-word pairs.

The outline of the new algorithm is presented in Fig. 11.2. We did not include in this outline some details which are explained in the sequel.

Trivial cases (one or both operands are 0 or 1) are treated in the obvious fashion.

Handling of signs and lengths is similar to what happens in the addition algorithm. In fact, we will ignore the signs and treat the operands as *positive*, while the output is also positive.

The routine `ShiftOne`(X) shifts the least-significant bits out of the non-zero X. The routine `ShiftTwo`(X, Y) shifts out the *common* least-significant bits from its arguments (of which at least one must be nonzero). Both routines operate in (almost) constant time, because the probability that many least-significant *words* are null is very small. As for the *bits*, first the number of shift-positions are determined using the least-significant (pair of) word(s), then the shift is performed systolically in one step.

Both shifting routines work on *complement* representation, performing *arithmetic* shifts (that is, sign-bits are replicated).

$G \leftarrow$ `IntSysGCD`(A, B) [$C \leftarrow GCD(A, B)$]
    $(A, B) \leftarrow$ `ShiftTwo`(A, B) [shift common zeroes]
    `while` $B \neq 0$ [main loop]
        $A \leftarrow$ `ShiftOne`(A) [shift A]
        $B \leftarrow$ `ShiftOne`(B) [shift B]
        $(x, y, x', y', s) \leftarrow$ `Cofactors`(a_0, b_0) [compute cofactors]
        $A' \leftarrow$ `LinComb`(x, A, s, y, B) [first linear combination]
        $B' \leftarrow$ `LinComb`$(x', A, 1 - s, y', B)$ [second linear combination]
        `if` $A' \neq 0$ [replace]
            `then` $(A, B) \leftarrow (A', B')$
            `else` $(A, B) \leftarrow (B', A')$
    [end of main loop: B is 0, A is the GCD]
    $G \leftarrow$ `ComplIfNeg`(C) [complement G if negative]

Figure 11.2: Systolic multiprecision GCD computation.

The routine `LinComb`(x, X, s, y, Y) computes the linear combination $x * X \pm y * Y$ (parameter s indicates $+$ or $-$). The routine works for negative operands also, using complement representation. A mask is used in order to indicate the range of correct values.

In order to avoid rippling the carries at each step, X, Y and the result of the linear combination are represented by *two* arrays of values (that is, a pair of values in each processor), one array containing the actual digits, and one containing the carries which are not propagated yet. During each linear combination, the carries are propagated only 1 step, after which each carry becomes at most 1 (this decreases the cost of the next multiplication). This scheme insures that the linear combination is performed in constant time.

Note that the *nonredundant* radix representation is *not* fully computed - however the least-significant digit of the result is always correct, because the carries propagate in the opposite direction. Or, only the values of the least-significant digits are needed for the next reduction step. It is this particular detail which makes the generalization of the binary GCD algorithm suitable for systolic implementation, in contrast with the Lehmer-Euclid algorithm.

After the main loop, G is complemented if negative, and its length is found by the same procedure as in the addition algorithm. In fact, G should be also right-shifted with the same number of binary positions which were shifted out from the inputs at the beginning. However, in the actual implementation we perform `ShiftTwo(...)` *before* calling the GCD routine, thus the normalization is still correctly done. Indeed, the GCD computation is needed for the *normalization* of the rational fraction E'/F'. We shift out of E', F' the common trailing binary zeroes, obtaining $E", F"$. Then the GCD algorithm is used to find $G" = GCD(E", F")$, and then $E = E"/G"$ and $F = F"/G"$ are found by exact division. Note that $G"$ is always odd, which suits well the needs of the exact division algorithm.

The main reduction scheme works only if the operands are multiprecision. If the GCD is single precision, then at some moment both operands A, B might also become single precision. From this moment the Euclidean algorithm is used for finding the GCD.

11.4 Exact division

The final stage of computation consists in performing the exact divisions by the GCD. As explained in the previous section, the divisor is already odd, hence the exact division algorithm introduced in chapter 4 can be applied without any preprocessing. In chapter 5 several systolic variants of this algorithm are described. We choose for implementation version 1 (see section 5.3.3), which suits well the particular characteristics of this application. Namely, the operations are simpler because global communication can be used (global communication is cheap on MasPar), and also the digits of the result are pushed out *during* the computation - hence allowing overlapping of computation and I/O operations.

The details of the algorithm are presented in Fig. 11.3.

The vector B in this algorithm is external to the processor array, and it is filled-up by the external environment with the digits supplied step by step by the systolic algorithm (lines [4] and [5] in fig. 11.3).

The function `ModInv`(a) is based on the recursion developed in section 5.1, hence we avoid the (expensive) extended Euclidean algorithm.

A simplified version of the function `LinComb` from the GCD algorithm is used for performing the operation $C - a' * A$. Again the carries are not propagated at each step, because only the correct value of the least-significant digit of C is needed for continuing the computation.

The handling of signs and lengths of the operands and result (not detailed in the outline) is done in a straight forward manner, using the principles

$B \leftarrow$ `IntSysEDIV`(C, A) [$B \leftarrow C/A$]
  $a' \leftarrow$ `ModInv`(a) [find ${a_0}^{-1} \bmod 2^{32}$]
  `while` $C \neq 0$ [main loop]
    $b \leftarrow (c_0 * a') \bmod 2^{32}$ [find next digit of the quotient ...]
    $B_{\text{next}} \leftarrow b$ [...and push it out]
    $C \leftarrow$ `LinComb`$(1, C, 1, a', A)$ [$C \leftarrow C - a' * A$]
    `for` $i = 0, 1, \ldots$ `in parallel do` [shift C left]
      $c_i \leftarrow c_{i+1}$
  [end of main loop]

Figure 11.3: Systolic multiprecision exact division.

shown in the addition algorithm.

11.5 Experimental results

The algorithms were implemented on a computer MasPar MP-1 having an array of 32 by 32 processors. The programs handle this two-dimensional array as an one-dimensional array of $32 * 32$ processors, virtually connecting the rows at their edges. The implementation language is `mpl-cc`, a dialect of C enhanced with special constructs for handling SIMD parallelism (see section 10.3.1 for details).

We timed the execution for random inputs having length up to 100 32-bit words. That means GCD is computed for operands having (roughly) 300 words, while output is usually small (single precision). The timings are presented in fig. 11.4. The times consumed for addition and `ShiftTwo` before GCD computation are 0.70 and 0.35 milliseconds, respectively, and are not shown in the figure.

The most important characteristic of the timing is the *linear dependence* of the lengths of the input. This shows that the systolic model can be effectively used on MasPar architecture for implementing long integer arithmetic.

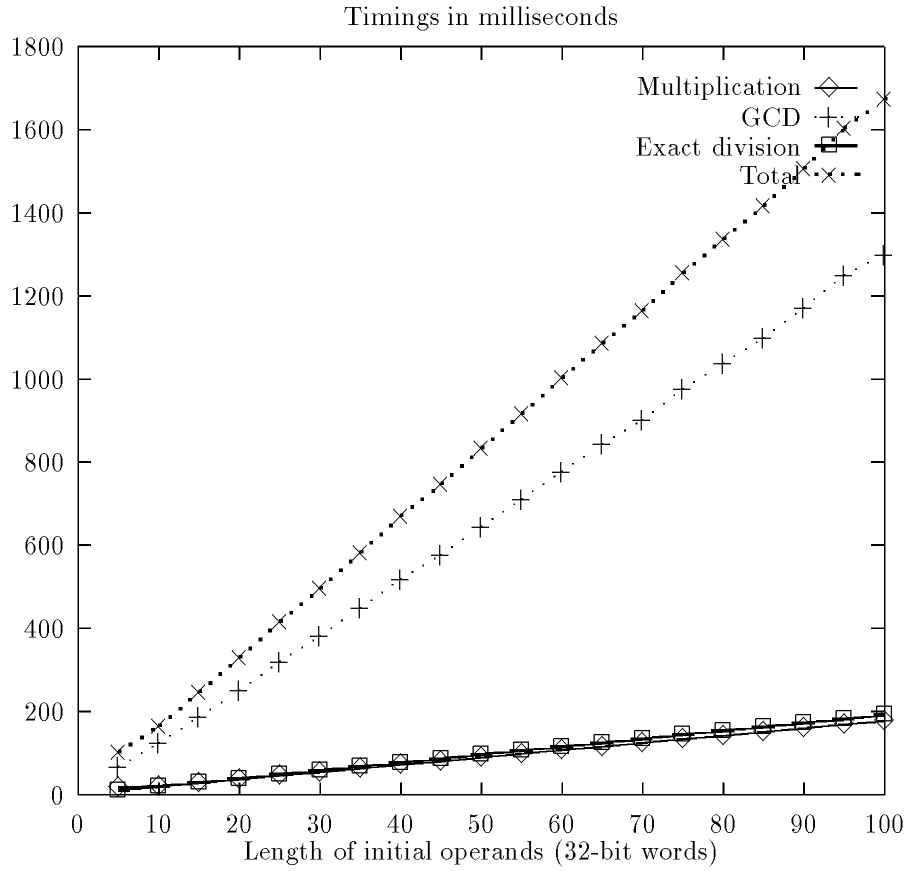

Figure 11.4: Timings of the rational reduction and its components.

Chapter 12

Hardware implementation

We implement in hardware, using an FPGA development system from Concurrent Logic, several least-significant digits first algorithms for arbitrary precision arithmetic: addition, multiplication, exact division and GCD computation. The least-significant digits first approach allows digit pipelining between the different stages of computation. Although redundant representation is not used, the systolic algorithms are with bounded fan-in and bounded fan-out, and therefore easily scalable to arbitrary length operands.

A preliminary estimation shows that the performance for 32 bit operands is comparable to that of a fast RISC processor, but for long operands a speed-up of one to two orders of magnitude can be achieved (50 times for 50 words operands).

12.1 Introduction

The final aim of the research presented in this chapter is to speed-up computer algebra systems by use of a dedicated coprocessor for arbitrary precision arithmetic. Namely, we study the possibility of implementing in hardware the systolic algorithms needed for operations with rational numbers: multiplication, addition, GCD (Greatest Common Divisor) and exact division (i. e. division with null remainder). Than we present a way of aggregating these algorithms in a pipelining manner, such that some operations overlap in time.

The use of systolic approach simplifies the designing the coprocessor and leads to robust solutions (see [Kung, 1982]). Also, the literature is abundant in systolic algorithms for long integer arithmetic (see citations at respective sections).

The particularity of our approach in contrast to others (see e. g. [Brent and Kung, 1985, Kornerup and Matula, 1987, Guyot *et al.*, 1987, Carter, 1989, Guyot and Kusumaputri, 1991, Guyot, 1991, Bouraoui and Guyot, 1992]) consists of:

- All the operations treat the operands in *least-significant digits first* (LSF) manner. We think that for long integer arithmetic this approach is simpler and more natural than most-significant digits first (MSF) used in *on-line* arithmetic (see [Schwartzlander, 1990a], Chapter 3), which might be better suited for fixed point arithmetic.

- Aggregation of different stages of computation is done in digit-pipelining manner. This allows overlapping of the execution time. Input and output is also pipelined in digit-serial fashion, overlapping in time with the computations. This also simplifies the connection with the host computer.

- Only bounded fan-in / fan-out is used: digits from partial computations, as well as computational decisions are pipelined along the systolic arrays. This is important for FPLA implementation because global communications lead to high consumption of space and time. Also, locality of communications makes the device easily scalable for arbitrary-length operands.

We present in this chapter a novel systolic algorithm for normalization of arbitrary length rational numbers, that is, computation of the GCD and of the reduced fraction. This algorithm is based on the "plus-minus" scheme introduced in [Brent and Kung, 1983]. In contrast with their algorithm and others (e. g. [Yun and Zhang, 1986], [Guyot, 1991]), the algorithm we present here has several new features:

- A mobile tag marks the sign-bit of the operands during GCD computation. This allows easier treatment of operands having different lengths, as well as easier detection of termination and of the GCD sign. In comparison with the systolic scheme of [Brent and Kung, 1985], which computes the GCD in $4n$ steps (worst case) using $4n$ cells of 24 bits, our algorithm needs $6n$ steps ($4n$ in the average) using n cells of only 8 bits. Note also that if N is the length of the array, then the older algorithm needs $4N$ steps for inputs of *any* length, because the operands are piped *through* the array. In contrast, the computing time of our algorithms depends only on the *actual* length of the inputs.

- The algorithm does not use global broadcasting, but only neighbor to neighbor communication. This allows the scaling of the device for arbitrary length operands without increase of the per–step computing time.

- The reduced fraction is computed using a systolic algorithm for *exact division* (introduced in [Jebelean, 1993d]) which is also fully pipelined and has time/space requirements proportional to the length of the quotient.

For experiments we use the FPGA development system from Concurrent Logic (CLi) (includes VLSI CAD from Viewlogic). After the logic design phase (before layout), a device accommodating 8 bit operands consumes 1,500 CLi cells, hence 6,000 cells would be necessary per word. The simulation runs with a cycle of 64 ns, which leads to average timings of $10\mu s$ for 32 bit operands and $0.5ms$ for 50 word operands. A C-language implementation of the same operations on a DECstation 5000/240 needs $14\mu s$ and $26ms$ respectively, hence the timings are in the same range for word operands, while for longer operands the speed-up would be in the range of one to two orders of magnitude.

12.2 Rational addition

We shall confine our discussion to *addition of rational numbers*, since the other operations are either similar or simpler. Given the rational numbers A/B and C/D we want to compute:

$$\frac{X}{Y} = \frac{A}{B} + \frac{C}{D},$$

where X/Y is normalized. This involves the steps:

$$E \longleftarrow A * D + B * C, \quad F \longleftarrow B * D,$$

$$G \longleftarrow GCD(E, F),$$

$$X \longleftarrow E/G, \quad Y \longleftarrow F/G.$$

(We do not use here the ideas of [Henrici, 1956] because they do not increase efficiency in the case of digit-systolic parallel computations.)

The flow of computation, which coincides with the structure of our device, is shown in Fig. 12.1. Each rectangle in this figure represents an array of identical simple processing elements, dedicated to a certain operation:

MUL multiplies two integers. The operands are input in a serial fashion, least significant digits first (LSB) through the right–end of the array. The product is delivered in parallel fashion at the low part of the array, but the least significant digits are ready earlier (bit n is available after n cycles).

ADD is an array of full-adders which delivers each bit of the sum in the same cycle in which the corresponding bits of the operands are available. Inputs and output flow in bit parallel fashion.

GCD computes the GCD of E, F, more exactly it right-shifts its operands with the number of common trailing zero bits (giving E_s and F_s), and computes $G = GCD(E_s, F_s)$. The inputs are fed in digit parallel fashion, but the computation starts as soon as the least-significant bits are available. The result is delivered also in parallel, after an average number of $4n$ steps.

EDIV performs the *exact* divisions E_s/G and F_s/G, thus delivering the final result X, Y. The inputs are taken in parallel manner, while the outputs are given digit–serially through the right-end processor. The least-significant digits of the results are ready earlier (the n^{th} pair of bits is available after n cycles). It follows that the number of cycles equals the actual length of the results.

The way the different stages of computation are *horizontally* shown in Fig. 12.1 should not drive the reader to think that the MUL arrays will be abutted in that way. In the actual implementation, the three multiplication arrays are merged in order to form a single array, such that the rightmost end of each of them is accessible at the rightmost end of the physical array. Hence, the systolic device will be composed of a "sandwich" of 4 systolic arrays. Physically, one or two chips connected vertically will contain a vertical section of this sandwich, such that by simple tiling a device of the desired length can be obtained.

12.3 Multiplication

The systolic multiplication algorithm is designed in such a manner that it fits well with the next stage – GCD computation (Addition is quite simple, and as we will see in the next section, it can be embedded within the last multiplication cycle). Because the way the operations are pipelined within the GCD array, this does not need all the digits of the operands to be present in the array at the beginning of the computation. Namely, it is only needed that the n^{th} pair of digits to be fed at step n. Therefore, we can design a systolic multiplication algorithm which can overlap in time with GCD computation:

- The inputs are fed in digit-serial fashion, least-significant first, to the rightmost processor of the multiplier.
- The computation proceeds as soon as the first input digits are available, and starts to deliver the output such as the n-th bit of the product is ready after the n-th step of the computation.

This algorithm is a modified version of the well-known parallel-serial mltiplier (see section 3.6, to which a special pipelining scheme is added in other to serialize the input which was parallel in the older algorithm. The structure of a basic cell is depicted in Fig.12.2. The following 7 registers are used:

- a, b hold the operands. a is pipelined leftward at unit speed, and b at half-speed, using an auxiliary register b'.

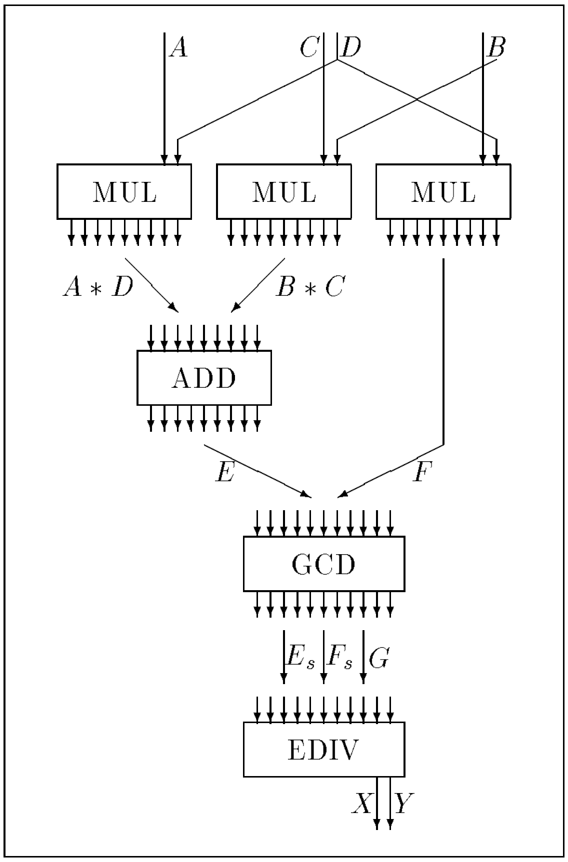

Figure 12.1: Structure of rational adder.

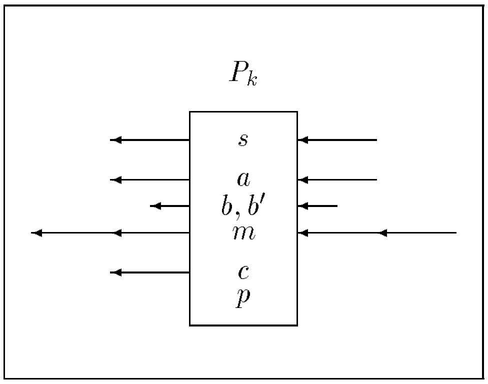

Figure 12.2: Structure of one systolic cell for multiplication.

- m moves b left at double-speed until it reaches the correct position to start participating in the computation.
- c, p hold the carry and the product. c is used by the left neighbor, p is kept steady and updated at each step.
- s holds the state information. While state is 0 the processors only send m to the left at double speed, and when state becomes 1 the processors start computing. s is pipelined leftward at unit speed.

We describe in the sequel the operation of each processor. By $v_{[-]}$ ($v_{[--]}$) we denote the value of the variable v on right (next to right) neighboring processor:

$a \leftarrow a_{[-]}$ [pipe a leftward]
`if` $s = 0$ [state ?]
    `then` $b \leftarrow m_{[--]}$
    `else` $b \leftarrow b'$
$b' \leftarrow b_{[-]}$ [pipe b at half-speed]
$(c, p) \leftarrow ab + c_{[-]} + p$ [update p]
$m \leftarrow m_{[--]}$ [pipe m at double speed]
$s \leftarrow s_{[-]}$ [pipe s]

Note that, in fact, this systolic algorithm will need $2n$ steps to compute the n-th digit of the product. The flow of computation is shown in fig. 12.3.

In order to synchronize it correctly with the GCD array, we implement two steps in one cycle, by replicating the transition function F (see Fig.12.4).

Three such multiplication arrays are used in the rational adder, for performing $A * D, \quad B * C, \quad B * D$.

12.4 Addition

Addition $A * D + B * C$ is performed by a simple string of full adders (FA in Fig.12.5) which deliver the sum of each pair of bits within the same cycle in which they are provided by the multiplier array. In order to insure correct synchronization, the carry is delayed through a latch (LA).

12.5 Systolic GCD computation

Parallelization of the classical Euclidean algorithm for integer GCD computation is difficult because of the carry propagation. Some of the difficulty is removed in the *binary* algorithm introduced in [Stein, 1967]. [Purdy, 1983] also designs a carry–free algorithm, whose worst-case time-complexity is still quadratic. In [Brent and Kung, 1983] and [Brent and Kung, 1985] an improvement of the binary algorithm (the *PlusMinus* algorithm) was introduced and parallelized in a systolic fashion, demonstrating that GCD computation in $O(n)$ steps is possible. [Yun and Zhang, 1986] suggest some further improvements of this last algorithm. The generalizations of the binary algorithm presented in [Jebelean, 1993a] and in [Sorenson, 1994] are also suitable for systolic parallelization, this time at the "word" level (i. e., digits in a high radix).

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

- The pipelining properties of the systolic algorithms allow efficient aggregation of several processing units. This is important when dealing with rational arithmetic, were several integer operations have to be

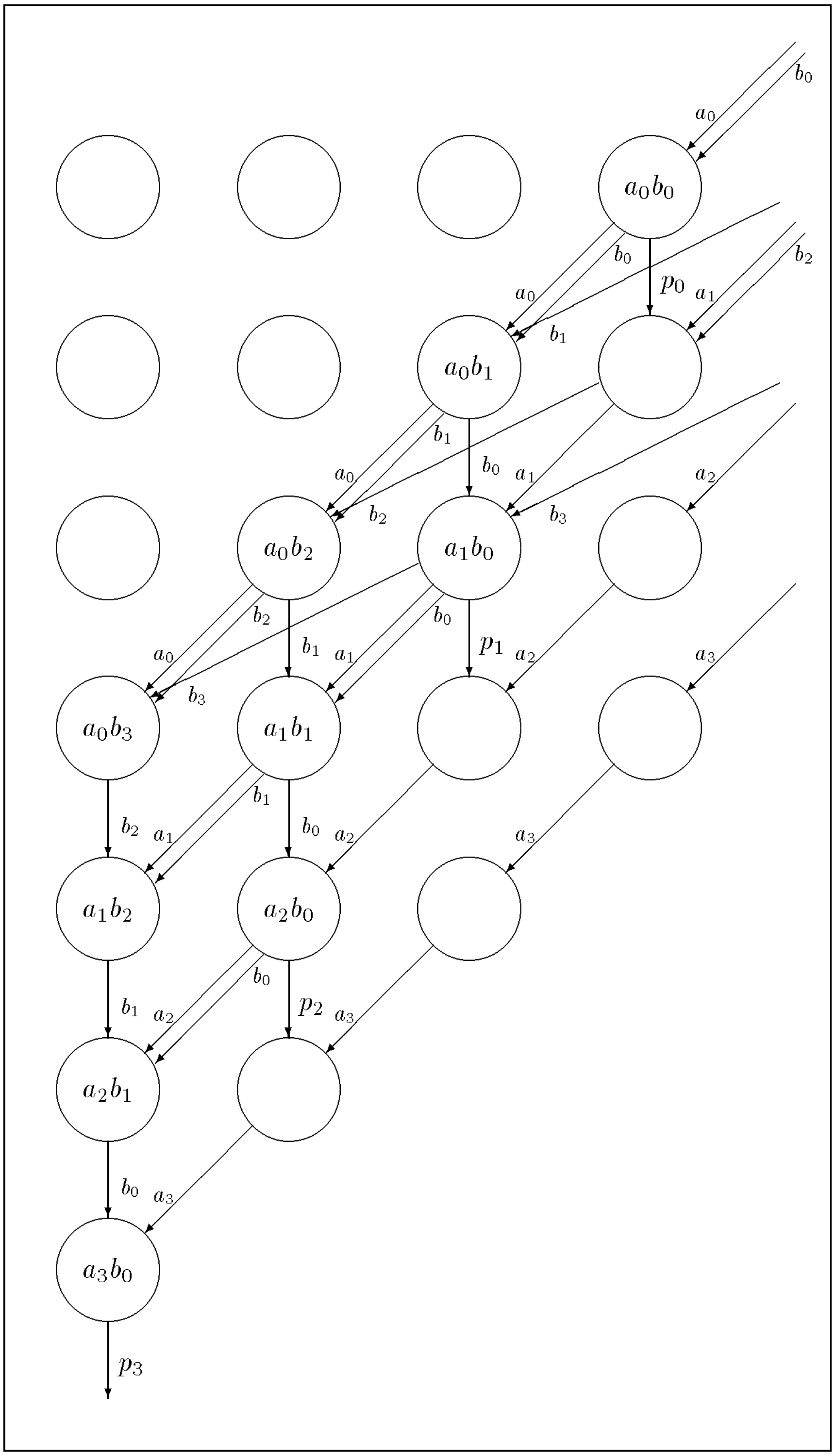

Figure 12.3: Data flow in systolic multiplication (carries not shown).

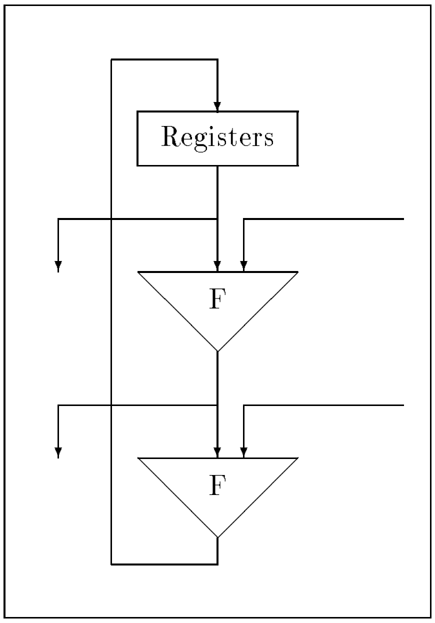

Figure 12.4: Implementing two multiplication steps in one cycle.

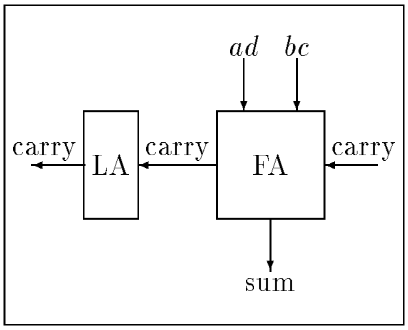

Figure 12.5: Processing element of the addition unit.

chained. We use here a variant of Brent–Kung algorithm ([Brent and Kung, 1983]), which works *least-significant digits first* (LSF), that is the same as Atrubins cellular algorithm for *multiplication* ([Atrubin, 1965]), and the same as the systolic *exact division* algorithm from [Jebelean, 1993d], based on [Jebelean, 1993b]. As for *addition* and *subtraction*, these are easy to realize in a LSF pipelined manner.

- In a fast communicating environment (as hardware implementation offers), there is little or no parallelization overhead in the systolic manner, while the overhead introduced by the asymptotically faster methods usually prevents the efficient implementation for small problems.

- The $O(n)$ speed–up given by systolic algorithms already leads to a significant improvement of the overall system. In fact, for multiple precision arithmetic performed on a "coprocessor" this is the best we can hope for, since the time needed to fetch the operands and send back the result to the host is already $O(n)$.

- The non-systolic algorithms lead to $O(n^2)$ area when implemented in hardware, which is not feasible for the range of operands we are interested in (up to 100 words of 32 bits). In contrast, systolic algorithms require only $O(n)$ hardware area.

- For the hardware implementation, the systolic approach offers the possibility of simple and regular design, which leads to a robust solution.

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

The theoretical basis of the algorithm are the relations:

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

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

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

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

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

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

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

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

12.6 Adapting systolic PlusMinus algorithm

We present here the modifications of Brent–Kung systolic GCD algorithm which make it suitable for our application. The original algorithm (see [Brent and Kung, 1983]) works by pipelining the n-digits operands *through* an array of $4n$ processors. In contrast, we choose a solution in which the operands are kept steady in an array of n processors, and they are shifted at different speeds according to the number of divisions by 2 which have to be performed. This solution is better suited to variable-length operands, because in this way one can ensure that the number of steps is proportional to the variable length of the inputs.

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

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

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

Note that the sign of an operand is given by the "tagged" bit (*plus* for *zero*, *minus* for *one*). Therefore, the use of mobile tags also leads to a way of detecting the sign of the resulting GCD: the value of the "tagged" bit is all the time propagated to the right, such that when the computation is ready the information about the sign is available in the rightmost processor. Since

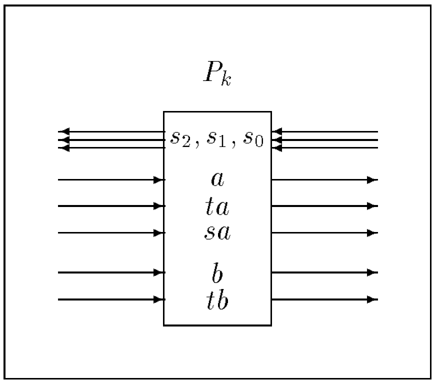

Figure 12.6: Structure of one systolic cell for GCD computation.

the pseudo-GCD is *odd*, changing the sign is done by inverting all the bits except the least-significant one. This requires only one cycle. Note also that in the context of rational normalization it is not needed to keep track of the initial common shifts and to left-shift the pseudo-GCD at the end. Rather, we save the values of the operands after the common shifts and we divide them by the pseudo-GCD.

A more detailed description of the algorithms follows.

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

Each processor has 8 one-bit registers (see also Fig.12.6):

- a and b hold the bits of the operands. These are initialized as described above. At the end of the computation a will hold the bits of the pseudo-GCD and b will be zero.

- ta and tb hold the tags of A and B. Initially the tags up to $n-1$ are zero and the tags from n on are one.
- sa holds the sign of A (zero for $+$, one for $-$), initially null.
- $s = (s_2, s_1, s_0)$ holds the "state" information, i. e. the code of the operation to be performed. There are 8 states:
  - **w**: wait;
  - **B**: shift both operands;
  - **C**: interchange and shift B;
  - **S**: shift B;
  - **P**: add, carry 1;
  - **p**: add, carry 0;
  - **M**: subtract, borrow 1;
  - **m**: subtract, borrow 0.

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

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

For describing the operations of the processors, we also use $v_{[+]}$ ($v_{[-]}$) to denote a value v on the left (right) neighboring processor.

The operation of processor P_0 looks as follows:

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

```
        then [interchange A and B and shift B]
            s ← C
            a ← b;        ta = tb
            b ← a[+];     tb ← ta[+]
    if a = 1 and b = 0
        then [shift B]
            s ← S
            b ← b[+];     tb ← tb[+]
    if a = 1 and b = 1
        then [plus or minus]
            if a[+] = b[+]
                then [minus]
                    s ← m ;   b ← 0
                else [plus]
                    s ← P ;   b ← 0
```

The rest of the processors operate like this:

```
if ta[+] = 1 [find sign of A]
    then sa ← a[+]
    else sa ← sa[+]
s ← s[-] [state is propagated leftward]
switch s[-] [right state is considered]
    case w  : [no modifications]
    case B  : [shift both A and B]
                a = a[+];     ta ← ta[+]
                b = b[+];     tb ← tb[+]
    case C  : [interchange A and B and shift B]
                a ← b;        ta = tb
                b ← a[+];     tb ← ta[+]
    case S  : [shift B]
                b ← b[+];     tb ← tb[+]
    case P , p : [plus]
        a ← b;        ta ← tb [set a to old b]
        (s2, b) ← a[+] + b[+] + s2 [set b to shifted sum]
        tb ← (ta AND tb) [tag the minimal correct position]
    case M , m : [plus]
```

$a \leftarrow b$; $\quad ta \leftarrow tb$ [set a to old b]
$(s_2, b) \leftarrow a_{[+]} - b_{[+]} - s_2$ [set b to shifted difference]
$tb \leftarrow (ta$ `AND` $tb)$ [tag the minimal correct position]

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

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

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

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

Also, at the end of GCD computation, the pseudo-GCD is complemented if *sa* in P_0 is 1. This requires only one cycle, because the pseudo-GCD is *odd.*

12.7 Exact division

By *exact division* we understand the division without remainder, i. e. when it is known in advance that the remainder is null. [Jebelean, 1993b] presents a new algorithm for exact division in high-radix, which speeds-up this operation by taking advantage of this knowledge. Also, this algorithm is suitable for systolic parallelization, because it works "right-to-left". A systolic exact division for binary-radix is contained in the division-with-remainder algorithm presented in [Purdy and Purdy, 1987], but this one uses global broadcasting. In [Jebelean, 1993d] we presented several LSF systolic algorithms for exact division, in high-radix, some of them without global broadcasting. We will use here one of these algorithms (see section 5.3.4), specialized for the binary-radix case and slightly modified in order to fit to the present problem. This algorithm accepts the inputs into the arrays, as they are provided by the previous pseudo-GCD algorithm, and produces the result (exact quotient) in a pipelined fashion, LSF, in the rightmost processor. The number of steps equals the length of the quotient.

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

Now if we denote by X' the shifted X:

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

then one has:

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

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

The systolic algorithm (see Fig.12.7) starts with the digits of A and X in the processors $P_0, \ldots, P_m$, where $m + 1$ is the length of the quotient. Processor P_0 has a distinct behavior: at step 0 it will output $x_0 = a_0$, and at step $k > 0$ it will output $x_k = a_{[+]} - g_1 * x_{k-1}$, where $a_{[+]}$ is the (updated) bit of A supplied by P_1 and g_1 is the one-before least-significant digit of G. The borrow obtained at each step is subtracted from x_k at the next step.

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

For more details concerning this algorithm the reader is referred to section 5.3.4.

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

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

The following registers are used by each processor:

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

The operation of processor P_0 is as follows:

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

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

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

`switch` s [four operation modes]

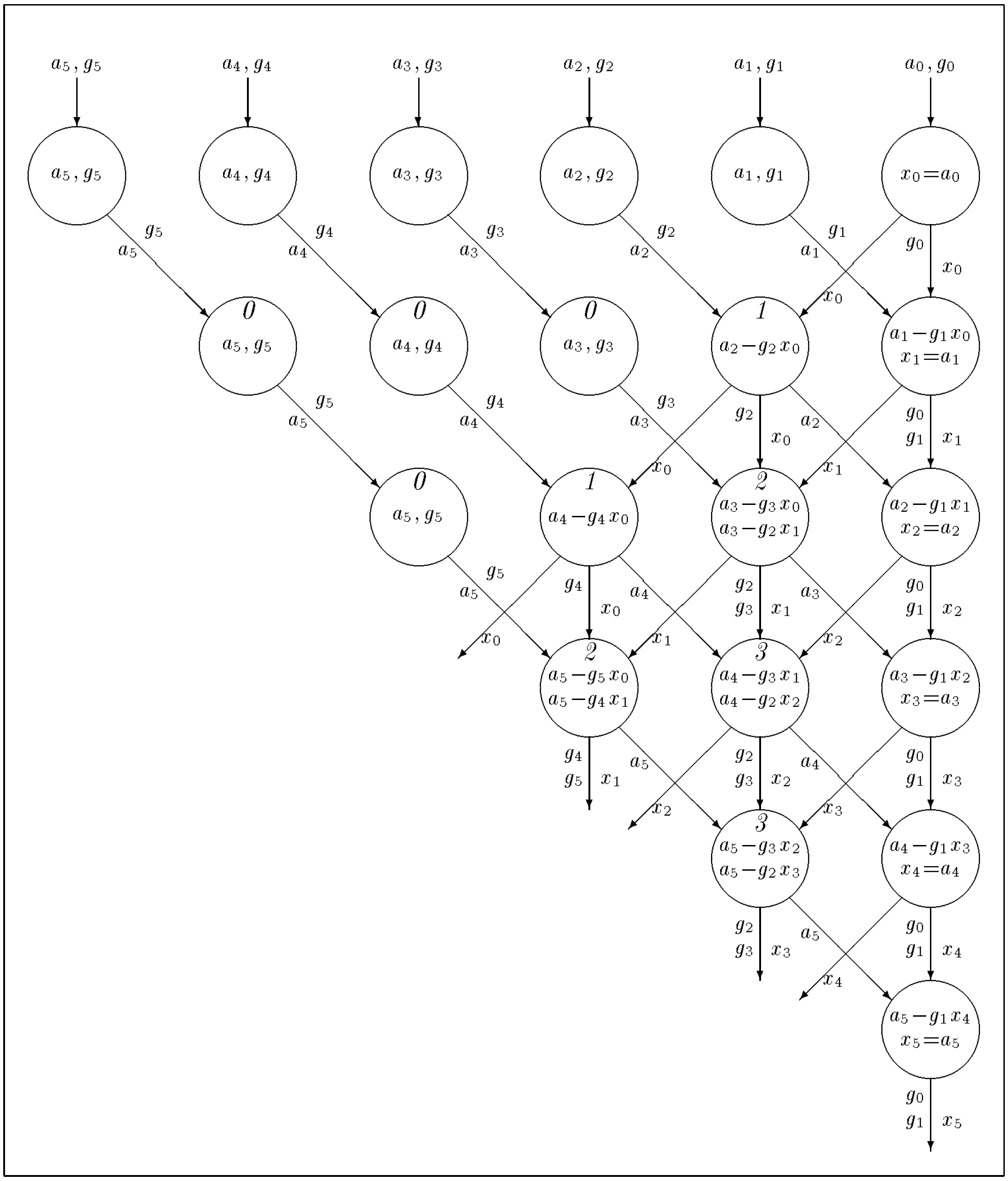

Figure 12.7: Data flow in systolic exact division (borrows not shown).

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

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

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

It is useful to note here that the performance of this counting component can be further improved by using the constant-time counter proposed by [Vuillemin, 1991].

12.8 Experiments and conclusions

We have implemented a software simulation of the systolic GCD algorithm. The number of steps for 1000 random inputs of 32 bits (GCD is small) was:

- average: 134.8 (4.2 n)
- maximal: 181
- minimal: 105

For 1000 random inputs of 32 bits with random GCD's of 16 bits the figures are:

- average: 90.5
- maximal: 117
- minimal: 65

Also, we carried out experiments using the FPGA development system from Concurrent Logic, Inc. (CLi)[1], with CAD system for circuit design from Viewlogic. Namely, we implemented and simulated a section of the systolic device big enough to handle 8 bits, that is, 4 bit inputs to multiplication, 8 bit inputs to GCD, (worst case) 8 bit outputs. After logic design only (before routing and layout on the programmable chip) the following data was observed:

- The multiplication arrays together with the addition need 500 CLi cells (3,133 equivalent gates).
- The GCD and exact division arrays need 1,002 cells (4,333 equivalent gates).
- All arrays can operate at a clock delay of 64 ns (speed 15 MHz). This timing corresponds to the delays through CLi6000 cells (see [Furtek, 1992]), but without routing.

[1]Since 1993 Concurrent Logic technology was transferred to the company AMTEL.

This means that a device for long rational arithmetic will need 6,000 cells per word, which could be probably accommodated in two CLi6006 chips.

At the (theoretical) speed above, a device accommodating 32 bits will operate in an average time of $5 * 32 * 64$ ns $= 10.2\mu s$. In comparison, a C-language implementation of rational addition runs on DECstation 5000/240 with an average of $14.3\mu s$, hence the two implementations are in the same range.

However, since the timing of the systolic device grows linearly with the input, for long operands a significant speed-up can be expected. The array will be able to normalize 50-word (of 32 bits) operands in about $5 * 50 * 32 * 64ns = 0.5ms$, while only GCD computation takes $26ms$ on a DECstation 5000/200 using the Lehmer-Euclid algorithm (see [Jebelean, 1993c] for timings of sequential GCD computation).

These, of course, are semi-theoretical projections. One would need to experiment with a real system in order to get the real figures, and one of the difficult problems which remains to be solved is the connection and synchronization of the coprocessor with the host. However, similar applications of FPGA devices for long integer multiplication in cryptography (see e. g. [Bertin *et al.*, 1989], [Shand *et al.*, 1991]) which were successful show that an efficient connection is feasible.

Chapter 13

Conclusions

The previous chapters demonstrate the possibility of organizing complex computations in such a way that they can be performed in systolic fashion. This is done for the particular area of *symbolic computation*, namely for the arithmetic of long integers, long rationals, and polynomials.

It is just the beginning of a long way. More research and experiments are needed in order to realize efficient implementations which could be actually used in computer algebra systems, giving a significant increase of performance.

The main conclusion of the thesis is that systolic parallelization is an effective and efficient approach to speeding-up long integer arithmetic by a significant factor. As a concrete result of our research, we develop systolic algorithms for arbitrary precision arithmetic which work least-significant digits first, in contrast with the traditional approach of on-line arithmetic. We introduce novel systolic algorithms for *exact division* and *GCD computation*, thus completing the collection of algorithms necessary for operating with rational numbers. Furthermore, all these algorithms operate in the same pipelining manner, hence an efficient aggregation of them becomes possible.

We present and compare several variants of these algorithms:

- with and without global broadcasting;
- at bit-level and at word-level.

Each variant is particularly suitable to a certain architecture, as we demonstrate by implementing them in hardware and on SIMD machine. An interesting implementation possibility remains still open: MIMD architecture at word-level (for instance, transputer network).

It is also important to note that our novel word-level algorithms (exact division, GCD computation) are faster than the previously known ones even in their sequential version.

However, the experiments presented in the thesis have the mere intention to argument the validity of our approach and to demonstrate the algorithms rather than to develop final software / hardware products that can be used in connection with computer algebra systems. A natural continuation of this research is to turn these experiments into “real-life” products.

Bibliography

[Abelson and Andreae, 1980] H. Abelson and P. Andreae. Information transfer and area-time tradeoffs for VLSI multiplication. *Communications of the ACM*, 23(1):20–23, January 1980.

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

[Aho *et al.*, 1974] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[Akl, 1989] S. G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, 1989.

[Akritas, 1989] A. G. Akritas. *Elements of Computer Algebra with Applications.* John Wiley, 1989.

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

[Aviziensis, 1961] A. Aviziensis. Signed-digit representation for fast parallel arithmetic. *IRE Trans. on El. Computers*, EC-10, 1961.

[Banerji and Kaushik, 1983] D. K. Banerji and S. Kaushik. Representation and processing of fractions in a residue number system. In T. R. N. Rao and P. Kornerup, editors, *SCA-6: 6th IEEE Symposium on Computer Arithmetic*, pages 29–36, Aarhus, Denmark, June 1983. IEEE Computer Society Press.

[Baudet *et al.*, 1983] G. M. Baudet, F. P. Preparata, and J. Vuillemin. Area-time optimal circuits for convolution. *IEEE Computer*, C-32(7):684–688, July 1983.

[Beame *et al.*, 1984] P. W. Beame, S. A. Cook, and H. J. Hoover. Log depth circuit for division and related problems. In *25th FOCS*, pages 1–6, 1984.

[Beardsworth, 1981] R. Beardsworth. On the application of array processors to symbol manipulation. In *SYMSAC'81*, 1981.

[Beister *et al.*, 1993] J. Beister, M. Kuhn, and R. Wollowski. High-level design and GAL implementation of an asynchronous controller for a daisy-chainable VME bus interrupt requester. In W. Moore and W. Luk, editors, *3^{nd} International Workshop on Field–Programmable Logic and Applications*, Oxford, UK, September 7 – 10, 1993. Jesus College.

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

[Bertin *et al.*, 1992] P. Bertin, D. Roncin, and J. Vuillemin. Programmable active memories: A performance assessment. Technical report, DIGITAL Paris Research Laboratory, February 1992.

[Booth, 1951] A. D. Booth. A signed binary multiplication technique. *Q. J. Mech. Appl. Math.*, 4:236–240, 1951.

[Bouraoui and Guyot, 1992] R. Bouraoui and A. Guyot. *EUCLIDES: An operator for the GCD and extended GCD of very large numbers*, page 32. TIMA-Laboratory, Univ. Grenoble, 1992. Research Project II.3.8, page 32.

[Boyle and Caviness, 1990] A. Boyle and B. F. Caviness, editors. *Future directions for research in symbolic computation.* Society for Industrial and Applied Mathematics, Philadelphia, 1990. Report to the National Science Foundation.

[Brent and Kung, 1981] R. P. Brent and H. T. Kung. The area-time complexity of binary multiplication. *Communications of the ACM*, 28(3):521–534, July 1981.

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

[Brent and Kung, 1985] R. P. Brent and H. T. Kung. A systolic algorithm for integer GCD computation. In K. Hwang, editor, *Procs. of the 7th Symp. on Computer Arithmetic*, pages 118–125. IEEE Computer Society, June 1985.

[Brent, 1976] R. P. Brent. Analysis of the binary Euclidean algorithm. In J. F. Traub, editor, *Algorithms and Complexity*, pages 321–355. Academic Press, 1976.

[Bromley *et al.*, 1988] K. Bromley, S.-Y. Kung, and E. Swartzlander, editors. *International Conference on Systolic Arrays*, San Diego, CA, May 1988. IEEE Computer Society Press.

[Bronstein, 1993] M. Bronstein, editor. *ISSAC'93: International Symposium on Symbolic and Algebraic Computation*, Kiev, Ukraine, July 1993. ACM Press.

[Brunvand, 1993] E. Brunvand. Windchime: An FPGA-based self-timed parallel processing. In W. Moore and W. Luk, editors, *3rd International Workshop on Field–Programmable Logic and Applications*, Oxford, UK, September 7 – 10, 1993.

[Bucci and Di Porto, 1988] M. Bucci and A. Di Porto. Fast serial-parallel multipliers. In T. Mora, editor, *AAECC-6: 6th International Conference on Applied Algebra, Algebraic Algorithms and Error Correcting Codes*, pages 111–121, Rome, Italy, 1988. Springer Verlag. LNCS 357.

[Buchberger *et al.*, 1982] B. Buchberger, G. E. Collins, and R. G. K. Loos (eds). *Computer Algebra, Symbolic and Algebraic Computation.* Springer Verlag, Wien-New York, 1982.

[Buchberger *et al.*, 1993] B. Buchberger, G. E. Collins, M. J. Encarnacion, H. Hong, J. R. Johnson, W. Krandick, R. Loos, A. M. Mandache, A. Neubacher, and H. Vielhaber. SACLIB 1.1 User's Guide. Technical Report 93–19, RISC–Linz, 1993.

[Buchberger, 1965] B. Buchberger. *An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-Dimensional Ideal (German).* PhD thesis, Univ. of Innsbruck, 1965. Ph.D. Thesis.

[Buchberger, 1985a] B. Buchberger. Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory. In Bose and Reidel, editors, *Recent* *trends in Multidimensional Systems*, pages 184–232, Dordrecht-Boston-Lancaster, 1985. D. Reidel Publishing Company.

[Buchberger, 1985b] B. Buchberger. The L-machine: An attempt at parallel hardware for symbolic computation. In J. Calmet, editor, *AAECC-3: 3rd International Conference on Applied Algebra, Algebraic Algorithms and Error Correcting Codes*, pages 333–347, Grenoble, France, July 1985. Springer Verlag. LNCS 229.

[Buchberger, 1990] B. Buchberger. An Implementation of Gröbner Bases in Mathematica. Technical Report 90–58, RISC–Linz, 1990.

[Carter, 1989] T. M. Carter. Cascade: Hardware for high/variable precision arithmetic. In *ARITH-9: 9th IEEE Symposium on Computer Arithmetic*, pages 184–191. IEEE Computer Society Press, 1989.

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

[Chu, 1962] Y. Chu. *Digital computer design fundamentals.* McGraw-Hill, New York, 1962.

[Cohen and van Gastel, 1992] A. M. Cohen and L. J. van Gastel, editors. *SCAFI'92: Studies in Computer Algebra for Industry II*, Amsterdam, 1992. Report Series of the Computer Algebra Netherlands Expertise Center.

[Cohen *et al.*, 1993] G. Cohen, T. Mora, and O. Moreno, editors. *AAECC-10: 10th International Conference on Applied Algebra, Algebraic Algorithms and Error Correcting Codes*, Puerto Rico, May 1993. Springer Verlag. LNCS 673.

[Cohen, 1991] A. M. Cohen, editor. *SCAFI'91: The 1991 Seminar on Studies in Computer Algebra for Industry*, Amsterdam, Dec. 1991. Report Series of the Computer Algebra Netherlands Expertise Center.

[Colagrossi and Limongelli, 1988] A. Colagrossi and C. Limongelli. Big numbers p-adic arithmetic: a parallel approach. In *AAECC-6*. Springer Verlag, 1988. LNCS 357.

[Collins and Encarnación, 1994] G. E. Collins and M. J. Encarnación. Efficient rational number reconstruction. Technical report, RISC-Linz, 1994. To appear.

[Collins and Loos, 1982] G. E. Collins and R. Loos. ALDES/SAC-2 now available. *ACM SIGSAM Bull.*, 1982.

[Collins, 1974] G. E. Collins. The computing time of the Euclidean algorithm. *SIAM Journal on Computing*, 3:1–10, 1974.

[Collins, 1975] G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *2nd GI Conference*, pages 134–183, Kaiserslautern, 1975. Springer Verlag. LNCS 33.

[Collins, 1980] G. E. Collins. Lecture notes on arithmetic algorithms, 1980. Univ. of Wisconsin.

[Cook, 1966] S. A. Cook. *On the minimum computation time of functions*. PhD thesis, Harvard University, May 1966.

[Dadda and Wah, 1993] L. Dadda and B. Wah, editors. *ASAP'93: Application Specific Array Processors*, Venice, Italy, October 1993. IEEE Computer Society Press.

[Danielsson, 1984] P. E. Danielsson. Serial/parallel convolvers. *IEEE Computer*, C-33(7):652–667, July 1984.

[Davenport and Robert, 1985] J. Davenport and Y. Robert. VLSI and computer algebra: the G.C.D. example. In Demongeot, Goles, and Tchuente, editors, *Dynamical systems and cellular automata*. Academic Press, 1985.

[Davenport *et al.*, 1988] J. H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra*. Academic Pres, 1988.

[Davis, 1985] P. J. Davis. Pompeiu's magic seven. In P. J. Davis and W. G. Chain, editors, 3.1416 and all that, pages 14 – 19. Birkhäuser, Boston, 2nd edition, 1985.

[Dongarra, 1993] J. J. Dongarra. Linpack benchmark results: highly parallel computing. *HPCwire*, October 1993. (electronic journal), item 2836.

[El-Desouky *et al.*, 1992] A. I. El-Desouky, M. M. Salem, A. O. A. El-Gwad, and L. M. Labib. A new technique for binary multiplier. *Int. J. of Mini and Microcomputers*, 14(2):68–76, 1992.

[Ercegovac, 1984] M. D. Ercegovac. On-line arithmetic: an overview. *SPIE*, 495 Real Time Signal Processing(VII):86–93, 1984.

[Fagin, 1992a] B. S. Fagin. Fast additon of large integers. *IEEE Computer*, 41:1069–1077, 1992.

[Fagin, 1992b] B. S. Fagin. Large integer multiplication on hypercubes. *J. of Parallel and Distr. Computing*, 14:426–430, 1992.

[Fortes and Wah, 1987] J. A. Fortes and B. W. Wah. Systolic arrays - From concept to implementation. *Computer*, pages 12–17, July 1987. (special issue on systolic arrays).

[Furtek, 1992] F. Furtek. An FPGA architecture for massively parallel computing. In *2nd International Workshop on FPLA*, Vienna, Austria, August 1992.

[Gex, 1971] A. Gex. Multiplier–divider cellular array. *Electronic Letters*, 7:442–444, 1971.

[Gibson and Gibbard, 1979] J. A. Gibson and R. W. Gibbard. Synthesis and comparison of two's complement multipliers. *IEEE Computer*, C-24:1020–1027, 1979.

[Goodman and McAuley, 1988] R. M. Goodman and A. J. McAuley. An efficient asynchronous multiplier. In K. Bromley, S.-Y. Kung, and E. Swartzlander, editors, *International Conference on Systolic Arrays*, pages 593–599, San Diego, CA, May 1988. IEEE Computer Society Press.

[Gosper, 1972] R. W. Gosper. Item 101 in HAKMEN. Technical Report AIM239, MIT, February 1972.

[Granlund, 1991] T. Granlund. GNU MP: The GNU multiple precision arithmetic library, 1991.

[Gregory and Krishnamurthy, 1984] T. Gregory and E. V. Krishnamurthy. *Methods and applications of error-free computation.* Springer-Verlag, 1984.

[Gruska, 1990] J. Gruska. Synthesis, structure and power of systolic computations. *Theoretical Computer Science*, 29(1):47–77, 1990.

[Guyot and Kusumaputri, 1991] A. Guyot and Y. Kusumaputri. OCAPI: A prototype for high precision arithmetic. In A. Halaas and P. B. Denyer, editors, *VLSI'91*, pages 11–18. IFIP, North Holland, 1991.

[Guyot *et al.*, 1987] A. Guyot, Y. Herreros, and J.-M. Muller. JANUS, an on-line multiplier/divider for manipulating large numbers. In M. J. Irwin and R. Stefanelli, editors, *ARITH-8: 8th IEEE Symposium on Computer Arithmetic*, pages 106–111, Como, Italy, May 1987. IEEE Computer Society Press.

[Guyot, 1991] A Guyot. OCAPI: Architecture of a VLSI coprocessor for the GCD and extended GCD of large numbers. In *ARITH-10: 10th IEEE Symposium on Computer Arithmetic*, Grenoble, France, June 1991. IEEE Computer Society Press.

[H. F. Mattson, 1991] T. R. N. Rao H. F. Mattson, T. Mora, editor. *AAECC-9: 9th International Conference on Applied Algebra, Algebraic Algorithms and Error Correcting Codes*, New Orleans, October 1991. Springer Verlag. LNCS 539.

[Habibi and Wintz, 1970] A. Habibi and P. A. Wintz. Fast multipliers. *IEEE Computer*, C-19(2):153–157, February 1970.

[Hennie, 1961] F. C. Hennie. *Iterative Arrays of Logical Circuits.* MIT Press, Cambridge, Mass, 1961.

[Henrici, 1956] P. Henrici. A subroutine for computations with rational numbers. *Journal of the ACM*, 3:6–9, 1956.

[Herman, 1969] G. T. Herman. The computing ability of a developmental model for filamentous organisms. *Journal of Theoretical Biology*, 25:421–435, 1969.

[Hwang and Briggs, 1984] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1984.

[Hwang, 1979] K. Hwang. Global and modular two's complement multiplier. *IEEE Computer*, C-28:300–306, April 1979.

[Irwin and Owens, 1992] M. J. Irwin and R. M. Owens. A micro-grained VLSI signal processor. In *ICASSP-92*, pages 641–644, March 1992.

[Jebelean, 1988] T. Jebelean. Cellular automata as parallel devices for language recognition. *Analele Univ. Timişoara, St. Matem.*, XXVI(3):29–45, 1988.

[Jebelean, 1993a] T. Jebelean. A generalization of the binary GCD algorithm. In M. Bronstein, editor, *ISSAC'93: International Symposium on Symbolic and Algebraic Computation*, pages 111–116, Kiev, Ukraine, July 1993. ACM Press.

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

[Jebelean, 1993c] T. Jebelean. Comparing several GCD algorithms. In *ARITH-11: IEEE Symposium on Computer Arithmetic*, pages 180–185, Windsor, Canada, June 1993.

[Jebelean, 1993d] T. Jebelean. Systolic algorithms for exact division. In *Workshop on Fine Grain and Massive Parallelism*, pages 40–50, Dresden, Germany, April 1993. Published in *Mitteilngen–Gesellschaft für Informatik e. V. Parallel Algorithmen und Rechnerstrukturen*, Nr. 12, July 1993.

[Kamal *et al.*, 1975] A. K. Kamal, H. Singh, and D. P. Agrawal. A generalized pipelined array. *IEEE Computer*, C-23:533–536, 1975.

[Kannan *et al.*, 1987] R. Kannan, G. Miller, and L. Rudolph. Sublinear parallel algorithm for computing the greatest common divisor of two integers. *SIAM Journal on Computing*, 16:7–16, 1987.

[Karatsuba and Ofman, 1962] A. Karatsuba and Yu Ofman. Multiplication of multidigit numbers on automata. *Sov. Phys. Dokl.*, 7:595–596, 1962.

[Katona and Legendi, 1981] E. Katona and T. Legendi. Cellular algorithms for fixed point decimal addition and multiplication. *EIK*, 17(11/12):637–644, 1981.

[Katona, 1982] E. Katona. Binary addition and multiplication in cellular space. *Acta Cybernetica*, 5:457–464, 1982.

[Knuth, 1981] D. E. Knuth. *The art of computer programming*, volume 2. Addison-Wesley, 2 edition, 1981.

[Kolář and Sasaki, 1992] M. Kolář and T. Sasaki. Multivariate quotient by power-series division. *SIGSAM Bulletin*, 26:17–20, 1992.

[Kornerup and Matula, 1987] P. Kornerup and D. W. Matula. A bit-serial arithmetic unit for rational arithmetic. In M. J. Irwin and R. Stefanelli, editors, *ARITH-8: 8th IEEE Symposium on Computer Arithmetic*, pages 204–211, Como, Italy, May 1987. IEEE Computer Society Press.

[Kronecker, 1901] L. Kronecker. *Vorlesungen über Zahlentheorie*, volume 1. Teubner, Leipzig, 1901.

[Kuechlin *et al.*, 1991] W. Kuechlin, D. Lutz, and N. Nevin. Integer multiplication on PARSAC-2 on stock microprocessors. In H. F. Mattson, T. Mora, and T. R. N. Rao, editors, *AAECC-9: 9th International Conference on Applied Algebra, Algebraic Algorithms and Error Correcting Codes*, New Orleans, October 1991. Springer Verlag. LNCS 539.

[Kung and Leiserson, 1978] H. T. Kung and C. E. Leiserson. Systolic arrays (for VLSI). In *Sparse Matrix Processing*, pages 256–282, Orlando, Fla., 1978. Academic Press.

[Kung and Leiserson, 1980] H. T. Kung and C. E. Leiserson. Systolic arrays (for VLSI). In C. Mead and L. Conway, editors, *Introduction to VLSI systems*, pages 271–292. Adisson-Wesley, Reading, Mass., 1980.

[Kung, 1981] H. T. Kung. Use of VLSI in algebraic computation: some suggestions. In *Proc. of 1981 ACM symp. on Symbolic and Algebraic Computation*, pages 218–222. ACM, 1981.

[Kung, 1982] H. T. Kung. Why systolic architectures? *Computer*, 15:37–46, 1982.

[Kung, 1988] S. Y. Kung. *VLSI Array Processors.* Prentice Hall, 1988.

[Kusche, 1991] K. Kusche. Parallel symbolic computation: pointers to the literature. Technical Report 91–59, Risc–Linz, December 1991.

[Lazard, 1992] D. Lazard. Stewart platform and Gröbner basis. In Parenti-Castelli and Lenarcic, editors, *Proc.* 3^{rd} *Int. Workshop on Robot Kinematics*, Ferrara, Sept., 1992.

[Lehmer, 1938] D. H. Lehmer. Euclid's algorithm for large numbers. *Am. Math. Mon.*, 45:227–233, 1938.

[Limongelli, 1993a] C. Limongelli. On an efficient algorithm for big rational number arithmetic by parallel p-adics. *Journal of Symbolic Computation*, 15(2), February 1993.

[Limongelli, 1993b] C. Limongelli. Rational number arithmetic by parallel p-adic algorithms. In *2nd ACPC Conference*, 1993.

[Lipson, 1981] J. D. Lipson. *Elements of computer algebra with applications.* Benjamin/Cummings, 1981.

[Luk and Vuillemin, 1983] W. K. Luk and J. E. Vuillemin. Recursive implementation of optimal-time VLSI integer multipliers. In Anceau and Aas, editors, *VLSI'83*, pages 155–162. Elsevier North–Holland, 1983.

[Mahler, 1973] K. Mahler. *Introduction to p-adic Numbers and Their Functions.* Cambridge University Press, 1973.

[MasPar90, 1990] MP–1 family data–parallel computers, 1990. MasPar Corp.

[MasPar92, 1992] MasPar data–parallel programming languages, 1992. MasPar Corp.

[McNaughton, 1961] R. McNaughton. The theory of automata, a survey. In *Advances in Computers*, volume 2, pages 379–421. Academic, New York, 1961.

[Mead and Conway, 1980] C. Mead and L. Conway. *Introduction to VLSI systems.* Adisson-Wesley, Reading, Mass., 1980.

[Moenck, 1973a] R. T. Moenck. Fast computation of GCDs. In *ACM Vth Symposium on Theory on Computing.* ACM, 1973.

[Moenck, 1973b] R. T. Moenck. Fast computation of GCDs. In *ACM* V^{th} *Symp. Theory of Computing*, pages 142–151. ACM, 1973.

[Moore and Luk, 1993] W. Moore and W. Luk, editors. *3rd International Workshop on Field–Programmable Logic and Applications*, Oxford, UK, September 7 – 10, 1993.

[Nagendra *et al.*, 1993] C. Nagendra, R. M. Owen, and M. J. Irwin. Digit systolic algorithms for fine-grain architectures. In *ASAP'93*, Venice, Italy, October 1993. Submitted.

[Neun and Melenk, 1990] W. Neun and H. Melenk. Very large Gröbner basis calculations. In Zippel, editor, *Computer algebra and parallelism. Proceedings of the second International Workshop on Parallel Algebraic Computation*, pages 89–100, Ithaca, May 1990. LNCS 584, Springer Verlag.

[Norton, 1989] G. H. Norton. Precise analyses of and the right- and left-shift greatest common divisor algorithms for $GF(q)[x]$. *SIAM Journal on Computing*, 18:608–624, 1989.

[Norton, 1990] G. Norton. On the asymptotic analysis of the Euclidean algorithm. *Journal of Symbolic Computation*, 10:53–58, 1990.

[Norton, 1992] G. H. Norton. Computing GCD by normalized division. *Aplicable Algebra in Engineering, Communication and Computing*, 2:275–295, 1992.

[Oldfield *et al.*, 1993] J. V. Oldfield, K. Shen, and C. J. Kapper. Genetic string comparison with a unidirectional algorithm implemented as a self-timed CAL array. In W. Moore and W. Luk, editors, *3^{nd} International Workshop on Field–Programmable Logic and Applications*, Oxford, UK, September 7 – 10, 1993. Jesus College.

[Peng and Hudson, 1988] S. Peng and T. Hudson. Parallel algorithms for multiplying very large integers. In *18th Int. Conf. on Parallel Processing*, volume 3, pages 173–177, 1988.

[Petkov, 1989] N. Petkov. *Systoliche Algorithmen und Arrays*. Akademie Verlag, Berlin, 1989.

[Planet *et al.*, 1993] P. Planet, G. Privat, and M. Renaudin. Asynchoronous relaxation of locally-coupled automata networks, with application to parallel VLSI implementation of iterative image processing algorithms. In L. Dadda and B. Wah, editors, *ASAP'93: Application Specific Array Processors*, pages 156–159, Venice, Italy, October 1993. IEEE Computer Society Press.

[Pompeiu, 1959] D. Pompeiu. *Oeuvre mathématique*. Académie de R. P. R., Bucarest, 1959.

[Ponder, 1988] C. G. Ponder. Parallel processors and systems for algebraic manipulation: current work. *ACM SIGSAM Bulletin*, 22(3):15–21, July 1988.

[POSSO, 1992] POSSO project (Polynomial Systems Solving) – ESPRIT III Basic Research Action 6846, 1992.

[Preparata and Vuillemin, 1981] F. P. Preparata and J. Vuillemin. Area-time optimal VLSI networks for computing integer multiplication and discrete Fourrier transform. In *ICALP'81*. Springer Verlag, 1981. LNCS 115.

[Preparata and Vuillemin, 1990] F. P. Preparata and J. E. Vuillemin. Practical cellular dividers. *IEEE Trans. on Computers*, C-39:605–614, 1990.

[Preston and Duff, 1984] K. Preston and M. J. B. Duff. *Modern Cellular Automata and Applications.* Plenum Press, New York, 1984.

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

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

[Quinton and Roberts, 1989] P. Quinton and Y. Roberts. *Algorithmes et Architectures Systoliques.* Masson, Paris, 1989.

[Roch, 1989] J. L. Roch. PAC: Towards a parallel computer algebra co-processor. In Della Dora and Fitch, editors, *Computer algebra and parallelism*, pages 33–50. Academic Press, 1989.

[Roch, 1990] J. L. Roch. An environment for parallel algebraic computation. In Zippel, editor, *Computer Algebra and Parallelism, Proceedings of the second International Workshop on Parallel Algebraic Computation*, pages 33–50, Ithaca, USA, May, 1990. LNCS 584, Springer Verlag.

[Schönhage and Strassen, 1971] A. Schönhage and V. Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7:281–292, 1971.

[Schönhage, 1966] A. Schönhage. Multiplikation grosser zahlen. *Computing*, 1:182–196, 1966.

[Schönhage, 1971] A. Schönhage. Schnelle Berechung von Kettenbruchentwicklugen. *Acta Informatica*, 1:139–144, 1971.

[Schönhage, 1980] A. Schönhage. Storage modification machines. *SIAM J. Comput.*, 9(3):490–508, August 1980.

[Schwartzlander, 1990a] E. E. Schwartzlander, editor. *Computer Arithmetic*, volume 2. IEEE Computer Society Press, 1990.

[Schwartzlander, 1990b] E. E. Schwartzlander, editor. *Computer Arithmetic*, volume 1. IEEE Computer Society Press, 2nd edition, 1990.

[Seidensticker, 1983] R.B. Seidensticker. Continued fractions for high-precision and high accuracy computer arithmetic. In T. R. N. Rao and P. Kornerup, editors, *SCA-6: 6th IEEE Symposium on Computer Arithmetic*, pages 184–193, Aarhus, Denmark, June 1983. IEEE Computer Society Press.

[Shallit and Sorenson, 1993] J. Shallit and J. Sorenson. Analysis of a left-shift binary algorithm. Submitted to J. of Symbolic Computation, 1993.

[Shand and Vuillemin, 1993] M. Shand and J. Vuillemin. Fast implementaton of RSA cryptography. In *ARITH-11: 11th IEEE Symposium on Computer Arithmetic*, pages 252–259, Windsor, Ontario, June 1993. IEEE Computer Society Press.

[Shand *et al.*, 1991] M. Shand, P. Bertin, and J. Vuillemin. Hardware speedups in long integer multiplication. *ACM SIGARCH*, 19:106–133, 1991.

[Soderstrand *et al.*, 1986] M. A. Soderstrand, W. K. Jenkins, G. A. Jullien, F. J. Taylor, and eds. *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing.* IEEE Press, New York, 1986.

[Sorenson, 1994] J. Sorenson. Two fast GCD algorithms. *J. of Algorithms*, 16:110–144, 1994.

[Spaniol, 1981] O. Spaniol. *Computer Arithmetic. Logic and Design.* John Willey & Sons, 1981.

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

[Stenzel *et al.*, 1977] W. J. Stenzel, W. J. Kubitz, and G. H. Garcia. A compact high-speed parallel multiplication scheme. *IEEE Computer*, C-26(10):948–957, 1977.

[Svoboda, 1962] A. Svoboda. The numerical system of residual classes. In W. Hoffman, editor, *Digital Information Processors.* John Wiley, 1962.

[Thompson, 1979] C. D. Thompson. Area-time complexity for VLSI. In *SIGACT: 11th Annual ACM Symp. on Theory of Computing*, pages 81–88, May 1979.

[Toffoli and Margolus, 1987] T. Toffoli and N. Margolus. *Cellular Automata Machines*. The MIT Press, Cambridge, Mass., 1987.

[Toom, 1963] A. L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics*, 3:714–716, 1963.

[Tošić and Stoićev, 1991] M. B. Tošić and M. K. Stoićev. Pipelined serial/parallel multiplier with contraflowing data streams. *Electronic Letters*, 27(25):2361–2363, 1991.

[Trivedi and Ercegovac, 1977] K. S. Trivedi and M. D. Ercegovac. On-line algorithms for division and multiplication. *IEEE Trans. on Computers*, C-26(7):681–687, 1977.

[von Neumann, 1951] J. von Neumann. The general logical theory of automata. In L. A. Jeffries, editor, *Cerebral Mechanisms in Behavior – The Hixon Symposium*, New York, 1951. John Wiley & Sons.

[von Neumann, 1966] J. von Neumann. The theory of self-reproducing automata. Urbana, 1966.

[Văcariu, 1992] 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), 1992. Patent Application 892/92, Vienna: Österreichisches Patentamt.

[Vuillemin, 1991] J. Vuillemin. Constant time arbitrary length synchronus binary counters. In *ARITH-10: 10th IEEE Symposium on Computer Arithmetic*, pages 180–183, Grenoble, France, June 1991. IEEE Computer Society Press.

[Vuillemin, 1993] J. Vuillemin. On circuits and numbers. Technical Report 25, Digital Paris Research Laboratory, 1993.

[Wang *et al.*, 1982] P. S. Wang, M. J. T. Guy, and J. H. Davenport. P-adic reconstruction of rational numbers. *ACM SIGSAM Bulletin*, 16(2):2–3, May 1982.

[Wang, 1981] P. S. Wang. A p-adic algorithm for univariate partial fractions. In *SYMSAC'81: ACM Symposium on Symbolic and Algebraic Computation*, pages 212–217. ACM, Inc., 1981.

[Wang, 1992] P. S. Wang, editor. *ISSAC'92: International Symposium on Symbolic and Algebraic Computation*, Berkeley, CA, July 1992. ACM Press.

[Weber, 1993] Ken Weber. The accelerated integer GCD algorithm. Technical report, Kent State University, 1993. To appear in ACM Trans. on Math. Software.

[Weeks, 1989] D. Weeks. Adaptation of SAC-1 algorithms for an SIMD machine. In J. Della Dora and J. Fitch, editors, *Computer Algebra and Parallelism*, pages 167–177. Academic Press, 1989.

[Windsteiger, 1990] W Windsteiger. An approach to object-oriented programming in c. Technical Report 90–57, RISC–Linz, 1990.

[Windsteiger, 1992] W. Windsteiger. Gröbner bases: A characterization by Syzygy completeness and an implementation. Technical Report 92–20, RISC–Linz, 1992.

[Yasura and Yajima, 1984] H. Yasura and S. Yajima. Hardware algorithms for VLSI systems. In T. L. Kunii, editor, *VLSI Engineering*. Springer Verlag, 1984. LNCS 163.

[Yun and Zhang, 1986] D. Y. Y. Yun and C. N. Zhang. A fast carry-free algorithm and hardware design for extended integer GCD computation. In *ACM SYMSAC'86*, pages 82–84. ACM, 1986.

[Yun, 1976] D. Y. Y. Yun. Algebraic algorithms using p-adic construction. In *ACM Symposium on Symbolic and Algebraic Computation*, 1976.

[Zassenhaus, 1981] H. Zassenhaus. Polynomial time factoring of integral polynomials. *ACM SIGSAM Bulletin*, 15:6–7, 1981.

[Zuras, 1993] D. Zuras. On squaring and multiplying large integers. In E. Swartzlander, M. J. Irwin, and G. Jullien, editors, *ARITH-11: 11th IEEE Symposium on Computer Arithmetic*, pages 260–271, Windsor, Ontario, June 1993. IEEE Computer Society Press.

Appendix A

Research papers related to the thesis

Most of the chapters of this thesis are based on previously published research papers:

Chapter 2: B. Buchberger, T. Jebelean — *Parallel rational arithmetic for Computer Algebra Systems: Motivating experiments*, RISC-Linz Report 92-29, May 1992; ACPC Workshop, Weinberg–Austria, April 1992.

Chapter 4: T. Jebelean — *An algorithm for exact division*, Journal of Symbolic Computation, 15 (2), February 1993, pp. 169–180.

Chapter 5: T. Jebelean — *Systolic Algorithms for Exact Division*, PARS 93 (Workshop on Fine Grain and Massive Parallelism, Dresda, Germany, April 1993). Published in *Mitteilungen–Gesellschaft für Informatik e. V. Parallel Algorithmen und Rechnerstrukturen*, Nr. 12, July 1993, pp. 40–50.

Chapter 6: T. Jebelean — *A Generalization of the Binary GCD Algorithm*, ISSAC'93 (International Symposium on Symbolic and Algebraic Computation, Kiev, Ukraine, July 1993), M. Bronstein (ed.), ACM Press, pp. 111–116.

Chapter 7: T. Jebelean — *Improving the multiprecision Euclidean algorithm*, DISCO'93 (Design and Implementation of Symbolic Computation Systems, Gmunden, Austria, September 1993). A. Miola (ed.), Springer Verlag LNCS 722, pp. 45–58,

Chapter 8: T. Jebelean — *Comparing Several GCD Algorithms*, ARITH-11 (11^{th} IEEE Symposium on Computer Arithmetic, Windsor, Ontario, June 1993), E. Schwartzlander, M. J. Irwin, G. Jullien (eds.), IEEE Computer Society Press, pp. 180–185.

Chapter 9: B. Buchberger, T. Jebelean — *On the possibility of implementing fine grain systolic algorithms on certain parallel architectures*, RISC–Linz Report 92-72, Dec 1992.

Chapter 10: T. Jebelean — *Systolic Multiplication on MasPar*, RISC–Linz Report 92–68, November 1992.

Chapter 11: T. Jebelean — *Systolic multiprecision arithmetic on MasPar*, International Seminar on Gröbner Bases and Related Topics, Dagstuhl, January 1994.

Chapter 12: T. Jebelean — *Systolic multiprecision arithmetic using FPGA*, FPLA'93: International Workshop on Field Programmable Logic and Applications, Oxford, UK, September 1993, W. Luk and W. Moore (eds), to appear.

Chapter 12: T. Jebelean — *Systolic Normalization of Rational Numbers*, ASAP 93 (Application Specific Array Processors, Venice, Italy, October 1993), L. Dadda and B. Wah (eds), pp. 502–513, IEEE Computer Society Press, pp. 502–513.

Furthermore, the author presented the ideas underlying this research at several international scientific events:

- B. Buchberger, T. Jebelean — *Systolic algorithms in Computer Algebra: State of the project*, Presented at the NATO ASI on Parallel Processing on Distributed Memory Multiprocessors, July 1–12, 1991, Ankara; RISC–Linz report 92–38.

- T. Jebelean — *Systolic Multiprecision Arithmetic*, Applicable Algebra Tagung, Oberwolfach, Germany, February 1993.

- B. Buchberger, T. Jebelean — *Systolic Multiprecision Arithmetic*, International Workshop on Parallel Processing in Education, Miskolc, March 1993.

Appendix B

Curriculum Vitae

B.1 Personal Data

Full Name:	Tudor Jebelean
Academic Degree:	M.Sc. (Dipl.-Ing.)
Birth Date:	December 1, 1955 (Timişoara, Romania)
Citizenship:	Romania
Personal Status:	Married, one daughter
Languages:	English (fluently), French (fluently), German (weak).

B.2 Education

- **Ph.D. Student of Technical Sciences** (since October 1990)
  Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria.

- **M.Sc. Degree in Computer Science with Distinction** (July 1979)
  Diploma Thesis: "SCREEN: Multiuser interactive system for remote data processing"; University of Timişoara, Romania.

- **Student in Computer Science** (September 1975 – July 1979)
  University of Timişoara, Romania.
  The study included: Mathematical Analysis, Algebra and Linear Programming, Geometry, Differential Equations, Mechanics, Numerical Analysis, Probability and Statistics, Operational Research, Model Theory, Computer Science Bases (4 semesters), Operating Systems, Economic

Systems Management, Information Systems for Economy, Formal Languages and Compiling Techniques, Data Structures and Data Bases, and Automata Theory.

- **High School Exam (Bacalaureat)** (June 1974)
  Gymnasium 1, Timişoara, Romania.

- **Classical Education** (September 1962 – June 1974)
  Gymnasium 3 and 1, Timişoara, Romania.

B.3 Professional Experience

- **Analyst-Programmer** (August 1979 – December 1979)
  Computing Center, U.V. Arad, Romania.

- **Analyst-Programmer** (January 1980 – August 1986)
  Computing Center, University of Timişoara, Romania.

- **Assistant-Professor** (since September 1986)
  (promoted **Lecturer** in September 1991)
  Department of Computer Science, University of Timişoara, Romania.

- **Research Assistant** (since October 1990)
  Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria.

B.4 Lecturing Experience

At the University of Timişoara (Romania), from 1985 to 1990, my educational activity was in Formal Languages and Compiling Techniques, and in Computer Science Bases (lectures, exercise classes, computer work classes, and 3 published tutorials).

At Johannes Kepler University, Linz, Austria:

- **Theoretical Foundations of Parallel Programming** (Summer Semester 1991, 1992, 1993)
  24 hours course.

- **Automatic Theorem Proving A** (Winter Semester 1992/1993)
  24 hours course.

- **Systolic Algorithms for Arithmetic** (Winter Semester 1993/1994) 24 hours course.

B.5 Refereeing activity

- **ACPC 91**
  1st International Conference of the Austrian Center for Parallel Computation (ACPC), Salzburg, Austria, September 29 – October 2, 1991.

- **ICALP 93**
  20th International Colloquium on Automata, Languages, and Programming July 1993, Lund, Sweden.

- **DISCO 93**
  International Conference on Design and Implementation of Symbolic Computation Systems, Gmunden, Austria, September 1993.

- **ASAP 93**
  International Conference on Application Specific Array Processors, Venice, Italy, October 1993.

- **Journal of Symbolic Computation**
  November 1993.

- **ACM Trans. on Math. Software**
  January 1994.

- **IEEE Transactions on Computers**
  Special issue on computer arithmetic, to appear in 1994.

- **Journal of VLSI Signal Processing**
  Special issue on application specific array processors, to appear in 1994.

B.6 Systems Experience

- **Working Experience** on Apollo workstations (Domain OS), DEC workstations (Ultrix), DEC Vax (VMS), IBM mainframes (VM), PCs (DOS) and Apples (MacOs).

- **Programming Languages:**

- **Imperative:** C, Pascal, Fortran, BASIC.
- **Object-Oriented:** C++.
- **Declarative:** Lisp, Prolog.
- **Parallel:** C-programming on MIMD shared memory (Sequent Symmetry), C-programming on SIMD distributed memory (MasPar).

B.7 Reports and Unrefereed Publications

1. T. Jebelean — *Categorii de spaţii afine (Categories of affine spaces)* – National Conference of Student Research, Craiova 1976

2. T. Jebelean — *Aspecte ale compilării reentrante şi conversaţionale pe Felix C-256 (Aspects of re-entrying and conversational compiling on Felix C-256)* – National Conference of Computing Centers from Academic Education, Braşov 1981

3. T. Jebelean — *Algoritmi de compilare bazaţi pe forma poloneză inversă (Compiling algorithms using reverse polish form)* – National Conference of Computing Centers from Academic Education, Gura Humorului 1984

4. T. Jebelean — *A theory on postfix notation* – preprint SIAN (Seminarul de Informatică şi Analiză Numerică, University of Timişoara) 18, Timişoara 1984

5. T. Jebelean — *Right to left pushdown recognition* – preprint SIAN 19, Timişoara 1984

6. T. Jebelean — *Semantics on a simple language* – preprint SIAN 21, Timişoara 1985

7. T. Jebelean — *Constant-time recognition of regular languages by one-way one-dimensional cellular automata* – International Conference "300 Jahre Mathematische Gesellschaft in Hamburg", Hamburg 1990

8. T. Jebelean — *Language Recognition by Nonpropagating Cellular Automata* – Res. Report 90-73, Technical University of Delft, Oct. 1990.

9. B. Buchberger, T. Jebelean — *Systolic algorithms in Computer Algebra: State of the project*, Presented at the NATO ASI on Parallel Processing on Distributed Memory Multiprocessors, July 1–12, 1991, Ankara; RISC–Linz report 92–38.

10. B. Buchberger, T. Jebelean — *Parallel rational arithmetic for Computer Algebra Systems: Motivating experiments*, RISC-Linz Report 92-29, May 1992; ACPC Workshop, Weinberg–Austria, April 1992.

11. T. Jebelean — *Systolic Multiplication on MasPar*, RISC–Linz Report 92–68, November 1992.

12. B. Buchberger, T. Jebelean — *On the possibility of implementing fine grain systolic algorithms on certain parallel architectures*, RISC–Linz Report 92-72, Dec 1992.

13. T. Jebelean — *Systolic Multiprecision Arithmetic*, Applicable Algebra Tagung, Oberwolfach, Germany, February 1993.

14. B. Buchberger, T. Jebelean — *Systolic Multiprecision Arithmetic*, International Workshop on Parallel Processing in Education, Miskolc, March 1993.

15. T. Jebelean — *Systolic multiprecision arithmetic on MasPar*, International Seminar on Gröbner Bases and Related Topics, Dagstuhl, January 1994.

B.8 Refereed Publications

1. T. Jebelean — *SCREEN: Multiuser interactive system for remote data processing*, Diploma Thesis, University of Timişoara, Romania, 1979.

2. T. Jebelean — *On semantics of compiling* – Ann. Univ. Timişoara, Math.Sci., XXV (1987), 3, pp. 49–68.

3. T. Jebelean — *Functorial translation* - Ann. Univ. Timişoara, Math.Sci., XXVI (1988), 3, pp. 47–52.

4. T. Jebelean — *Cellular automata as parallel devices for language recognition* – Ann. Univ. Timişoara, Math.Sci., XXVI (1988), 3, pp. 29–45.

5. T. Jebelean — *Real-time recognition of context-free languages by one-way one-dimensional cellular automata* – Romanian National Conference on Theoretical Computer Science (INFO IASI'89), Iaşi 1989.

6. T. Jebelean — *Bilinear automata* – Studii şi Cercetări Matematice, Tom 42 (1) 1990, 19–29.

7. T. Jebelean — *An algorithm for exact division*, Journal of Symbolic Computation, 15 (2), February 1993, pp. 169–180.

8. T. Jebelean — *Systolic Algorithms for Exact Division*, PARS 93 (Workshop on Fine Grain and Massive Parallelism, Dresda, Germany, April 1993). Published in *Mitteilungen–Gesellschaft für Informatik e. V. Parallel Algorithmen und Rechnerstrukturen*, Nr. 12, July 1993, pp. 40–50.

9. T. Jebelean — *Comparing Several GCD Algorithms*, ARITH-11 (11^{th} IEEE Symposium on Computer Arithmetic, Windsor, Ontario, June 1993), E. Schwartzlander, M. J. Irwin, G. Jullien (eds.), IEEE Computer Society Press, pp. 180–185.

10. T. Jebelean — *A Generalization of the Binary GCD Algorithm*, ISSAC'93 (International Symposium on Symbolic and Algebraic Computation, Kiev, Ukraine, July 1993), M. Bronstein (ed.), ACM Press, pp. 111–116.

11. T. Jebelean — Systolic multiprecision arithmetic using FPGA, FPLA'93: International Workshop on Field Programmable Logic and Applications, Oxford, UK, September 1993, W. Luk and W. Moore (eds), to appear.

12. T. Jebelean — *Improving the multiprecision Euclidean algorithm*, DISCO'93 (Design and Implementation of Symbolic Computation Systems, Gmunden, Austria, September 1993). A. Miola (ed.), Springer Verlag LNCS 722, pp. 45–58,

13. T. Jebelean — *Systolic Normalization of Rational Numbers*, ASAP 93 (Application Specific Array Processors, Venice, Italy, October 1993), L. Dadda and B. Wah (eds), pp. 502–513, IEEE Computer Society Press, pp. 502–513.

B.9 Research Interests

- Parallel Computation,
- Systolic Algorithms and Architectures,
- Symbolic and Algebraic Computation,
- Arbitrary Precision Arithmetic.