ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická

CZECH TECHNICAL UNIVERSITY IN PRAGUE
Faculty of Electrical Engineering

Modular System for Error-free Computation of Large and Ill-conditioned Set of Linear Equations

Modulární systém pro bezchybný výpočet velkých a špatně podmíněných soustav lineárních algebraických rovnic

Ing. Róbert Lórencz, CSc.

Habilitation lecture / Habilitační přednáška

June 1, 2005

Summary

This work deals with the issue of design of a modular system for error-free computation of a Set of Linear Equations (SLE) based on the residual number system.

In the introduction, reasons and motivation of the chosen approach to the solution of the problem of an error-free computation of an ill-conditioned SLE are mentioned.

In the first part of the presentation, the mathematical principles and algorithms of an error-free solution of SLE on the residual number system base are introduced. A brief description of a modular system architecture is given. The architecture benefits from parallelism coming from the use of the residual number system. This architecture consists of identical residual processors (RP) implemented in hardware. Each of these processors solves a set of linear congruences (SLC) derived from SLE in modular arithmetic with its own modulo. The architecture of residual processors is designed with the aim to achieve the best performance of computation of SLC according to the Gauss-Jordan elimination. This architecture is covered by Czechoslovak patent [16]. One of the crucial points of the design of RP are its vector arithmetic units. The arithmetic units are also hardware identical and they perform four basic arithmetic operations in modular arithmetic. The performance of the whole computation of a SLE strongly depends on efficient execution of these basic arithmetic operations. The operation of multiplicative modular inverse has the greatest computational complexity. For this reason, the major emphasis of the design of the whole system is put on the computation of modular inverse.

The second part of this presentation describes a new algorithm for computing multiplicative modular inverse and its hardware architecture, which are covered by a Czech patent [12]. The presented new "Left-shift" algorithm has been designed with respect to efficient computation of modular inverse. Almost the least number of addition and subtraction operations, which represent a significant extension of the critical path of hardware implementation, is achieved. The permission of occurrences of negative values, which are represented in the two's complement code, has completely eliminated complex comparison tests (greater/lesser) during the calculation of modular inverse. In conclusion of the second part, a mathematical proof of correct calculation of modular inverse according to the "Left-shift" algorithm is introduced.

Finally, the overview of main achievements related to the design of a modular system for error-free computation of SLE and multiplicative modular inverse are summarized.

Souhrn

Tato práce se zabývá návrhem modulárního systému pro přesné řešení soustav lineárních rovnic (SLR) na bázi systému kódů zbytkových tříd.

V úvodní části jsou stručně popsány důvody a motivace zvoleného přístupu řešení problematiky přesného počítání špatně podmíněných SLR.

V první části prezentace jsou uvedeny základní matematické principy a algoritmy přesného řešení SLR na bázi systému kódů zbytkových tříd. Je zde popsána stručně architektura modulárního systému využívajícího paralelizmu plynoucího z použití systému kódů zbytkových tříd. Tato architektura se skládá z hardwarově identických residualních procesorů (RP). Každý z těchto procesorů řeší soustavu lineárních kongruencí (SLK) odvozenou ze SLR v dané modulární aritmetice s vlastním modulem. Architektura reziduálních procesorů je navržena tak, aby co nejefektivněji řešila SLK pomocí Gauss-Jordanovy eliminace. Tato architektura je předmětem československého patentu [16]. Jedním z klíčových bodů návrhu RP jsou jeho vektorově pracující aritmetické jednotky. Aritmetické jednotky jsou opět hardwarově identické a vykonávají čtyři základní aritmetické operace v modulární aritmetice. Od efektivity provádění těchto operací značně závisí efektivita celého výpočtu SLR. Výpočetně nejsložitější je operace multiplikativní modulární inverze. Z tohoto důvodu je při návrhu celého systému věnována této operaci velká pozornost.

V druhé části prezentace je uveden nový algoritmus výpočtu multiplikativní modulární inverze a jeho hardwarová architektura, které jsou předmětem českého patentu [12]. Prezentovaný nový "Left-shift" algoritmus je navržen s ohledem na co nejefektivnější výpočet modulární inverze. Zvlášť' velký důraz je kladen na co nejnižší počet operací sčítání a odčítání, které představují významné prodloužení kritické cesty hardwarové implementace. Povolením výskytu záporných čísel, reprezentovaných v doplňkovém kódu, jsou v průběhu výpočtu úplně odstraněny složité testy větší/menší. Na závěr druhé části je uveden matematický důkaz správnosti výpočtu multiplikativní modulární inverze počítané podle nového "Left-shift" algoritmu.

V závěru jsou shrnuty hlavní dosažené výsledky související s návrhem modulárního systému pro přesné řešení SLR a multiplikativní modulární inverze.

Keywords:

set of linear equations, multiplicative modular inverse, residual number system, modular arithmetic, Euclid algorithm, error-free computation, ill-conditioned problems, floating point arithmetic

Klíčová slova:

soustava lineárních rovnic, multiplikativní modulární inverze, systém kódů zbytkových tříd, modulární aritmetika, Euklidův algoritmus, bezchybné počítání, špatně podmíněné úlohy, aritmetika s plovoucí řádovou čárkou

Contents

1 Introduction **6**

2 Error-free Computation of a Set of Linear Equations **6**

2.1 Method of Solving a Set of Linear Equations using the Residual Number System . . . 6

2.2 Modular System for Solving Sets of Linear Equations Exactly . . . 9

3 Algorithm for Modular Inverse **11**

3.1 New Left-shift Algorithm for the Classical Modular Inverse . . . 11

3.2 Results and Discussion . . . 12

3.3 HW Implementation . . . 14

3.4 The Mathematical Proof of the Left-shift Algorithm . . . 15

4 Conclusion **18**

References **18**

Ing. Róbert Lórencz, CSc. **20**

1 Introduction

Many numerical methods were developed trying, with more or less success, to minimize the influence of the rounding errors on the resulting solution. One of the important study fields of numerical mathematics are the methods of error-free computation that use finite number systems, in which the computation is performed without rounding errors and which are used in digital computers. Consequently, any effort to use a digital computer to perform arithmetic in the field of real numbers often results in a failure because of the impossibility to map an infinite set of real numbers to a finite set of numbers represented inside a computer.

Usually we attempt to approximate the arithmetic of real numbers on a computer using the so-called floating-point numbers, more appropriately called the set of *computer-representable numbers*. The properties of the set of floating-point numbers and the rules for their representation in computers are contained in [1]. It is appropriate to mention here that on a binary computer the only candidates for membership in the floating-point number set are rational numbers of the form p/q, where p, q are integers, and q is a power of two. Thus, there is no possibility to represent the continuum of real numbers in any detail. The representation of a real number in a binary computer is provided by its closest computer-representable number, thereby introducing the rounding error. It is not difficult to show that an approximation using the finite set of floating-point numbers can cause incorrect and unusable results in the cases of ill-conditioned and numerically unstable tasks [7, 20]. This fact is a strong reason for using number systems in which we can perform exact arithmetic, i.e. without rounding errors. A residual number system can fulfill this requirement.

2 Error-free Computation of a Set of Linear Equations

The numerical tasks of linear algebra are often performed in a number system which eliminates partially or completely the rounding errors that occur during the calculation of the solutions [5, 8, 10, 17, 18, 23]. This is a motivation for a construction of straightforward algorithms for linear algebra that utilize primarily only basic arithmetic operations, which are easily implemented in arithmetic of various number systems. The next reason is the fact that solving problems of linear algebra is done frequently in numerical mathematics in particular it is the solution of linear algebraic equations. When solving a set of linear equations (SLE), one often meets the problem of an ill-conditioned matrix of the set. For large dense sets of linear equations, which are usually ill-conditioned, the stability of a numerical solution cannot be ensured. Rounding errors committed during the numerical computations involved in obtaining the solution of the problem cannot be tolerated. Many numerical methods were developed trying, with more or less success, to minimize the influence of the rounding errors on the resulting solution [4, 17, 21, 23].

2.1 Method of Solving a Set of Linear Equations using the Residual Number System

Residual number system (RNS) appears to be a suitable number system for implementing certain numerical methods for error-free solving of a set of linear equations [13]. One of

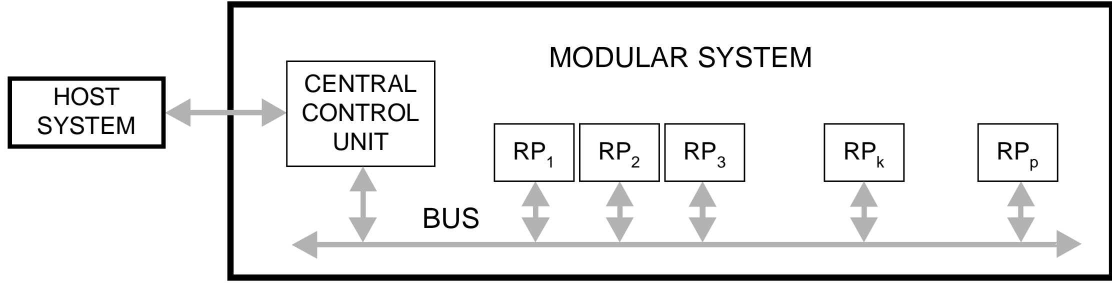

Figure 1: Modular system

such methods is described in [14, 15]. The computational complexity of this method is larger than the complexity of the original task of solving the a SLE in the set of floating-point numbers. Therefore, the mentioned papers describe a design of a parallel architecture of a modular system which speeds up the complete calculation. The method in papers [14, 15] describes the algorithms for the parallel architecture of a modular system for solving SLR exactly using RNS (see Figure 1), where RP_1, RP_2, …, RP_p are residual processors (p is number of processors), which are interconnected via BUSES. The principles of this approach are briefly described in the following text.

Let us have a SLE with integer coefficients

$$\mathbf{A}\mathbf{x} = \mathbf{b}. \tag{1}$$

If we denote $\mathcal{M}(\mathbf{b}) = \max(|b_i|)$ and $\mathcal{M}(\mathbf{A}) = \max(|a_{i,j}|)$, where $i, j \in [1, n]$, then the size of a modulus M of single-modulus residue arithmetic used in an error-free algorithm to solve (1) is given as [18]

$$M > 2\max\{n^{\frac{n}{2}} \cdot \mathcal{M}(\mathbf{A})^n,\ n(n-1)^{\frac{n-1}{2}} \cdot \mathcal{M}(\mathbf{A})^{n-1} \cdot \mathcal{M}(\mathbf{b})\}. \tag{2}$$

Since the modulus M estimated using (2) is very large, its application to calculations is not practical. The hardware architecture would be impossible to implement, or, in the case of a software realization, the computational complexity would be immense. Therefore it is more suitable to use a known set of moduli $m_1, m_2, \ldots, m_r$ of RNS using the multiple-modulus residual arithmetic. Then the calculation in each single-modulus residual arithmetic with modulus m_i can be carried out in parallel (in residue processors RP_1, RP_2, …, RP_p, where p can be different from r – see also Figure 1), as long as there is a dedicated hardware architecture that allows it. The moduli must fulfill the following conditions [5, 18]

a) $\displaystyle\prod_{i=1}^{r} m_i \geq M$,

b) each m_i for $i \in [1, r]$ is prime and

c) $m_1 < m_2 < \cdots m_r < (2^e - 1)$,

where e is the length of the word in bits (number of bits of internal data buses, registers and functional units of R_p).

Since the elements of the solution vector $\mathbf{x}$ are rational numbers in general, it is essential to utilize the following trick to solve a SLE with integer coefficients using RNS

$$\mathbf{A}\mathbf{z} = d\mathbf{b}, \tag{3}$$

where $d = \det \mathbf{A}$. The elements of the solution vector $\mathbf{z}$ of such a SLE are integeres, and then the solution vector $\mathbf{x}$ can be obtained using floating-point arithmetic:

$$\mathbf{x} = \frac{1}{d}\mathbf{z}. \tag{4}$$

The method of solving (1) according to the previous conditions consists of three basic steps [15]:

1. Conversion of the coefficients $a_{i,j}$ of matrix $\mathbf{A}$ and the coefficients b_i of vector $\mathbf{b}$, where $i, j \in [1, n]$, of SLE (1) to r systems of a set of linear congruences (SLC) $\mathbf{A}\mathbf{y}_{\mathrm{k}} \equiv \mathbf{b} \pmod{m_k}$, $k \in [1, r]$.

2. Solving r systems of SLC $\mathbf{A}\mathbf{y}_{\mathrm{k}} \equiv \mathbf{b} \pmod{m_k}$ using the Gaussian elimination with pivoting [15] in modular arithmetic modulo m_k, where $k \in [1, r]$. Also the determinants $d_k = \det \mathbf{A} \bmod m_k$ are computed. Then the solution of (3) modulo m_k is obtained as $\mathbf{z}_k = d_k \mathbf{y}_k$ for $k = 1, 2, \ldots, r$.

3. Conversion of the resulting vectors $\mathbf{z}_k$ and the determinants d_k using the mixed-radix conversion (MRC) algorithm [5] from the RNS into a single vector $\mathbf{z}$ such that $\mathbf{z}_k \equiv \mathbf{z} \pmod{m_k}$ for $k = 1, 2, \ldots, r$. The solution of (1) is obtained as $\mathbf{x} = \frac{1}{d}\mathbf{z}$.

The first step of the method can be executed in parallel using a certain broadcast technique. This means that each processor RP_k, $k \in [1, p]$, reads integer (or rational number) elements of the matrix $\mathbf{A}$ and the vector $\mathbf{b}$ of SLE, and converts them simultaneously into residues modulo $m_1, m_2, \ldots, m_r$. Assuming the number of processors p is equal to r, for the first step of the method we have $O(n^2)$ parallel operations that convert integers into residuals.

The second step of the method is completely parallel since the solutions of each SLC $\mathbf{A}\mathbf{y}_k \equiv \mathbf{b} \pmod{m_k}$ are independent for every $k = 1, 2, \ldots, r$. In the case when the number of processors p in a parallel system is equal to r, the processor k can be allocated for computations modulo m_k: processor k solves the SLC $\mathbf{A}\mathbf{y}_k \equiv \mathbf{b} \pmod{m_k}$, and computes the determinant $d_k = \det \mathbf{A} \bmod m_k$, and then proceeds to compute $\mathbf{z} = d_k\ \mathbf{y}_k \bmod m_k$. This computation is performed simultaneously by all processors $1, 2, \ldots, r$. Since the Gaussian elimination on a matrix of dimension n requires $O(n^3)$ arithmetic steps, assuming $p = r$ we get $O(n^3 p/r) = O(n^3)$ arithmetic steps for the second step of the method.

At the end of the second step of the method we obtain an n-element vector $\mathbf{z}_k$ and an integer d_k in processor k for $k = 1, 2, \ldots, r$. We apply the MRC algorithm [5] to compute an n-element vector $\mathbf{z}$ and an integer d. Let the $(n+1)$-element vector $\mathbf{t}_k$ be

$$\mathbf{t}_k = \begin{bmatrix} \mathbf{z}_k \\ d_k \end{bmatrix}. \tag{5}$$

We can use the MRC algorithm to compute the $(n+1)$-element vector $\mathbf{t}$ so that $\mathbf{t} \equiv \mathbf{t}_k \pmod{m_k}$ [24]. The MRC algorithm returns the vector $\mathbf{t}$, from which we extract the value of determinant d and the elements of vector $\mathbf{z}$. Then the final solution vector $\mathbf{x}$ is computed using (4).

The MRC sequential algorithm consists of two steps. First, the values $\mathbf{t}_k$ are updated according to the following recursion:

for $l := 1$ to $r-1$ do
for $k := l+1$ to r do
$\mathbf{t}_k := (\mathbf{t}_k - \mathbf{t}_l)c_{k,l} \bmod m_l,$

where $c_{k,l}$ are the multiplicative inverses of m_k modulo m_l for $1 \le k < l \le r$. After the above update process, the elements of the vectors $\mathbf{t}_k$ are the mixed-radix coefficients of the elements of the final vector $\mathbf{t}$, which can be obtained by computing

$$\mathbf{t} = \mathbf{t}_1 + m_1\mathbf{t}_2 + m_1m_2\mathbf{t}_3 + m_1m_2m_3\mathbf{t}_4 + \cdots + m_1m_2\cdots m_{r-1}\mathbf{t}_r.$$

Since the MRC algorithm of r integers takes $O(r^2)$ arithmetic operations [10, 18], we have $O(nr^2)$ arithmetic operations, which are required by sequential MRC algorithm for the $(n+1)$ vector $\mathbf{t}$.

In general, $p \le r$ processors are available. Hence, we partition the moduli set in such a way that each processor works with at most q moduli, where $q = \lceil r/p \rceil$.

Then the first step of the method will have $O(qn^2)$ operations of integer to residue conversion, and the second step requires $O(qn^3)$ modular arithmetic operations. In the case of $p \le r$ processors, the number of steps in the third step of the method depends on the parallel organization of the processors.

2.2 Modular System for Solving Sets of Linear Equations Exactly

The method of solving SLEs described in the previous part can be implemented in a special parallel hardware system which was published by the author in [14, 15].

The first of these papers [15] describes the method, the algorithm, and the corresponding parallel hardware architecture of a system for solving a dense SLE precisely. The error-free calculation of SLE with operations performed in residue arithmetic is implemented in this special modular system. The modular system has typically a parallel SIMD architecture, and consists of one control unit and several identical processing units – Residual Processors (RPs) RP_1, RP_2, …, RP_p interconnected with BUS (see Figure 1). The architecture of RP_k depicted in Figure 2 consists of Memory, Arithmetic Units AU_1, AU_2, …, AU_j … AU_{n+2} and the Control unit. Memory contains elements of matrix $\mathbf{A}$ and vector $\mathbf{y}$. Storing values of one row of the matrix from registers of AU's is performed bitwise via the Serial Inputs SI_1, SI_2, …, SI_{j-1}, …, SI_{n+1}. Loading values of rows of Memory to AU's is done via the Serial Outputs SO_2, SO_3, …, SO_j, …, SO_{n+1}. The bits of values of elements of the first column of the matrix are read by the Control unit via the parallel bus PO_1. All AU's and the Control unit are interconnected via Internal Bus. The described architecture of RP_k can solve SLC $\mathbf{A}\mathbf{y}_k \equiv \mathbf{b} \pmod{m_k}$ according to the second step on page 8. The residual processors RP_k, $k \in [1, p]$ together with the Control unit of the Modular system also support the execution of all conversion operations from integer to RNS and vice versa according to steps one and three described on page 8. The INV and DET units compute the modular multiplicative inverse and the determinant of $\mathbf{A}$, respectively.

The second paper [14] deals with the hardware architecture of individual parts of the modular system. Also, the spatial complexity of the individual parts and the whole modular system is introduced.

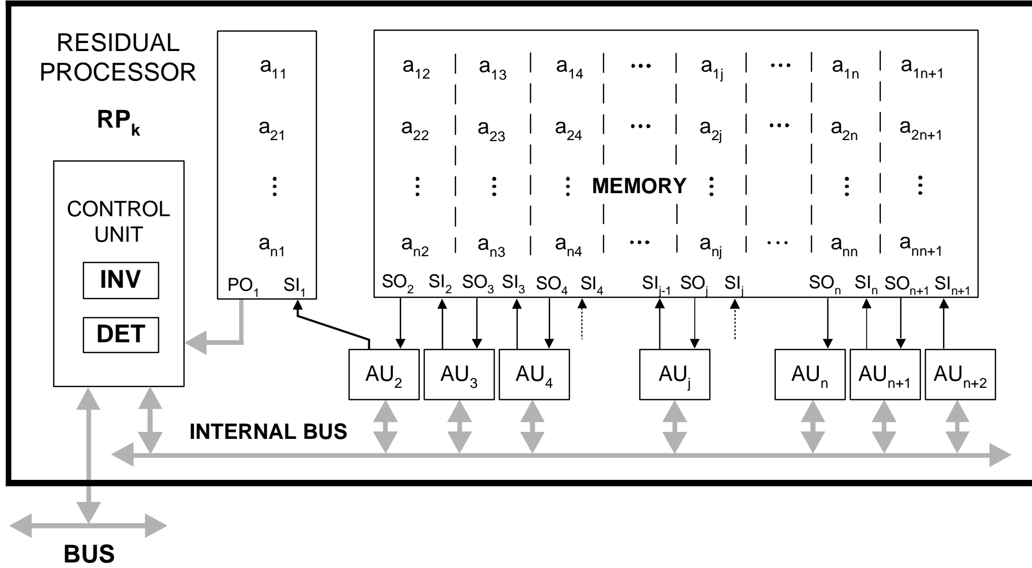

Figure 2: Residual processor

The main contributions of both papers are:

- The design of a new hardware architecture of a residual processor (Figure 2 of [15] and Figure 1). The interconnection of memory and arithmetic units covered by a patent [16] (see Figure 2). The architecture of RPs lower the time complexities of the conversion from SLE to SLCs and the SLCs computation by a factor of n.
- An efficient solution of a problem with negative values of the resulting vector.
- A discussion and calculation of the probability of a failure in the situation when $\gcd(M, d) > 1$, i.e. $d_k \equiv 0 \pmod{m_k}$ for any $k \in [1, r]$.
- An introduction of a hardware solution to the so-called "non-zero residue pivotization" of the Gauss-Jordan elimination of SLC.
- A detailed design of the hardware architectures of individual parts of the modular system with respect to an efficient execution of basic arithmetic operations in modular arithmetic.

The method of solving a SLE implemented in the modular system uses only four basic arithmetic operations, namely addition, subtraction, multiplication and division, to obtain the solution vector. For solving the SLC using RP's, four basic arithmetic operations executed in modular arithmetic are used. An important operation in modular arithmetic is multiplicative modular inverse because of its largest computational complexity. The modular system has a very efficient hardware implementation of addition, subtraction and multiplication. However, multiplicative modular inverse is realized by a look-up table (ROM), where for each integer in Galois Field GF(m_k), a value of the multiplicative inverse modulo m_k is associated. Such a solution has several drawbacks. First, with a growing modulus m_k the time complexity of the conversion also grows because larger addresses need to be decoded. The capacity of the ROM grows with the size of m_k exponentially. For some parameters of a SLE to be solved using the modular system the look-up table takes more space than the residual processor. Due to this fact it was needed to solve the problem of efficient computation of the multiplicative inverse (INV unit of RP, see Figure 2) in Galois Field GF(p), where p is prime, and its implementation in hardware.

3 Algorithm for Modular Inverse

Hardware implementation of the multiplicative modular inverse, in contrast to other basic modular operations, namely addition, subtraction, and multiplication, has not been fully explored yet due to the computational complexity of the modular inverse. There are many hardware applications in the area of cryptography and error-free numerical algorithms that utilize the computation of modular multiplicative inverse. Due to these facts, the design of new and efficient algorithms for computing the modular inverse suitable for hardware implementation on various platforms is always a current task in the field of computer science. Many different approaches for computing the modular inverse were developed, both for software and hardware implementations [2, 3, 5, 6, 9, 10, 22]. One of the well-known algorithms for computing the modular inverse is based on the Extended Euclidean Algorithm (EEA). The next section deals with the new efficient algorithms for computing modular inverse based on EEA and their implementation in hardware.

3.1 New Left-shift Algorithm for the Classical Modular Inverse

Using the classical definition, the multiplicative inverse of an integer $a \in [1, p-1]$ modulo the prime number p is define as integer $x \in [1, p-1]$ satisfying the condition $ax \equiv 1 \pmod{p}$. Often, the notation $x = a^{-1} \bmod p$ is used.

The new approach to the calculation of modular inverse [11, 12], which is the subject of this section, avoids the drawbacks of current algorithms, that are Penk's algorithm [10] and Kaliski's algorithm for Montgomery inversion [9]. In Penk's algorithm, these are especially: high number of tests, which essentially represent a subtraction and also an addition if some immediate results are negative. In case of the modular inverse calculation using the Montgomery modular inverse according to Kaliski, it is necessary to perform the deferred halving in k iterations in second phase of algorithm, including corrections of r if it is odd. An algorithm that avoids the mentioned problems is presented below:

LEFT-SHIFT ALGORITHM

Input: $a \in [1, p-1]$ and p
Output: $r \in [1, p-1]$, where $r = a^{-1} \pmod{p}$, c_u, c_v
  and $0 < c_v + c_u \leq 2n$
1. $u := p, v := a, r := 0, s := 1$
2. $c_u = 0,\ c_v = 0$
3. while$(u \neq \pm 2^{c_u}\ \&\ v \neq \pm 2^{c_v})$
4.   if $(u_n, u_{n-1} = 0)$ *or* $(u_n, u_{n-1} = 1\ \&\ \mathrm{OR}(u_{n-2}, \ldots, u_0) = 1)$ then
5.   if $(c_u \geq c_v)$ then
6. $u := 2u,\ r := 2r,\ c_u := c_u + 1$
7.   else
8. $u := 2u,\ s := s/2,\ c_u := c_u + 1$
9.   else if $(v_n, v_{n-1} = 0)$ *or* $(v_n, v_{n-1} = 1\ \&\ \mathrm{OR}(v_{n-2}, \ldots, v_0) = 1)$ then
10.   if $(c_v \geq c_u)$ then
11. $v := 2v,\ s := 2s,\ c_v := c_v + 1$
12.   else
13. $v := 2v,\ r := r/2,\ c_v := c_v + 1$
14.   else
15.   if $(v_n = u_n)$ then

16. $oper = " - "$
17. else
18. $oper = " + "$
19. if ($c_u \leq c_v$) then
20. $u := u$ *oper* v, $r := r$ *oper* s
21. else
22. $v := v$ *oper* u, $s := s$ *oper* r
23. if ($v = \pm 2^{c_v}$) then
24. $r := s$, $u_n := v_n$
25. if ($u_n = 1$) then
26. if ($r < 0$) then
27. $r := -r$
28. else
29. $r := p - r$
30. if ($r < 0$) then
31. $r := r + p$
32. return r, c_u, and c_v.

The Left-shift algorithm was designed to be easily implemented in hardware (see section 3.3). Registers Ru, Rv, Rs are $m = n + 1$ bit wide registers and contain individual values of the variables u, v, s. The value of variable r is in $m + 1$ bit wide register Rr. Counters Cu and Cv are auxiliary $e = \lceil \log_2 n \rceil$ bit wide counters containing values c_u and c_v. The presented Left-shifting binary algorithm computes the modular inverse of a according to equation $x = a^{-1} \bmod p$ using EEA and shifting the values u and v to the left, that is multiplying them by two. The multiplication is performed as long as the original value multiplied by 2^i is preserved, where i is the number of left shifts. Negative values are represented in the two's complement code. The shift is performed as long as the bits u_n, u_{n-1} or v_n, v_{n-1} are zeros for positive values or ones for negative values, while at least one of the bits u_{n-2}, u_{n-3}, $\ldots u_0$ or v_{n-2}, v_{n-3}, $\ldots v_0$ is not zero - binary 'OR' (steps 4 and 9). With each shift, counters Cu and Cv (values c_u and c_v) that track the number of shifts in Ru, Rv are incremented (steps 6, 8, 11, and 13). Registers Rr and Rs (values r and s) are also shifted to the right (steps 8 and 13) or left (steps 6 and 11) according to conditions in steps 5 and 10. In step 15, addition or subtraction, given variable *oper*, is selected according to sign bits u_n and v_n for the subsequent reduction of u, v and r, s in steps 20 and 22. Results of these operations are stored either in Ru and Rr (values u and r), if the number of shifts in Ru is less or equal to the number of shifts in Rv, or in registers Rv and Rs (values v and s) otherwise. The loop ends whenever '1' or '-1' shifted by the appropriate number of bits to the left appears in register Ru or Rv. Branch conditions used in steps 4, 9, and 15 are easily implemented in hardware. Similarly, the test in steps 5, 10, and 19 can be implemented by an e bit comparator of values c_u and c_v with two auxiliary single-bit flips-flops u/$\bar{\text{v}}$ and wu (see Figure 4).

3.2 Results and Discussion

A simulation and a quantitative analysis of the number of additions or subtractions ('+/−'), shifts and tests was performed for Left-shift algorithm, Kaliski's algorithm and Penk's algorithm. Simulation of modular inverse computation was performed for all integers $a \in [2, p-1]$ and all 1899 prime moduli $p < 2^{14}$ ($n \leq 14$) . A total of 14,580,841 inverses were computed by each method. Simulation results are presented in Table 1.

The number of all tests, additions and subtractions are listed in column "$+/-/tests$". The tests include all "greater than" and "less than" comparisons except '$v > 0$' in the main loop, which is essentially a '$v \neq 0$' test that does not require a subtraction. The "$+/-$" column lists additions and subtractions without tests. The column "$total\ shifts$" indicates the number of all shift operations during the computation. The last column lists the number of shifts minus the number of '$+/-$' operations, assuming the shift is performed together with storing the result of the '$+/-$' operation. The columns give minimum and maximum numbers of operations ($min - max$) and their average ($avg.$) values. Shift operations are faster in hardware than additions, subtractions, and comparison op-

Table 1: Results for primes less than 2^{14}

Algorithm	$+/-/tests$		$+/-$		$total\ shifts$		$shifts-(+/-)$	
	min - max	*avg.*	*min - max*	*avg.*	*min - max*	*avg.*	*min - max*	*avg.*
Left-shift	-	-	2 - 21	9.9	2 - 26	23.3	1 - 24	13.4
Kaliski's	5 - 45	26.2	4 - 40	21.1	6 - 54	38.2	0 - 43	17.1
Penk's	9 - 80	40.4	6 - 53	27.1	2 - 26	18.1	0	0

erations performed with the Ru, Rv, Rr, Rs registers. The comparison operations are about as slow as additions/subtractions, since they cannot be performed in parallel; they depend on data from previous operations. If a suitable code is used to represent negative numbers, this condition can be realized as a simple sign test, avoiding the complicated testing. Both Penk's algorithm and Kaliski's algorithm suffer from a large number of additions and subtractions that correct odd numbers before halving in Penk's algorithm and in Kaliski's algorithm, and convert negative numbers in Penk's algorithm. Moreover, Kaliski's algorithm needs twice the number of shifts compared to Penk's algorithm, required by the second phase.

The previous analysis shows that Left-shift algorithm removes drawbacks of Penk's algorithm and Kaliski's algorithm. Let us assume full hardware support for each algorithm and simultaneous execution of operations specified on the same line of the pseudocodes. Let us further assume that no test is needed in Kaliski's algorithm. Then, we can use the values in columns "$+/-$" and "$total\ shifts$" to compare the number of operations. Left-shift algorithm needs half the number of '$+/-$' operations compared to Kaliski's algorithm, and 2.7 times less the number of '$+/-$' operations compared to Penk's algorithm.

The simulation results from Table 1 for prime moduli less than 2^{14} ($n = 14$) are plotted in Figure 3. It shows the average number of cycles T needed to compute the modular inverse using all three algorithms as a function of the ratio ρ, where ρ is defined as the ratio of the critical path length in cycles of the shift and the critical path length in cycles of the adder/subtracter. All (Penk's algorithm) or a part of (Kaliski's algorithm and Left-shift algorithm) shift operations are included in '$+/-$' operations. Shift operations that are not performed as a part of '$+/-$' operations are performed individually (they are listed in the last column of Table 1).

With an increasing word length, the time complexity of shift operations remains constant. However, the complexity of additions/subtractions increases approximately $\lceil \log_2 m \rceil$ - times, m is the number of bits of a word. For long words, often used in cryptographic

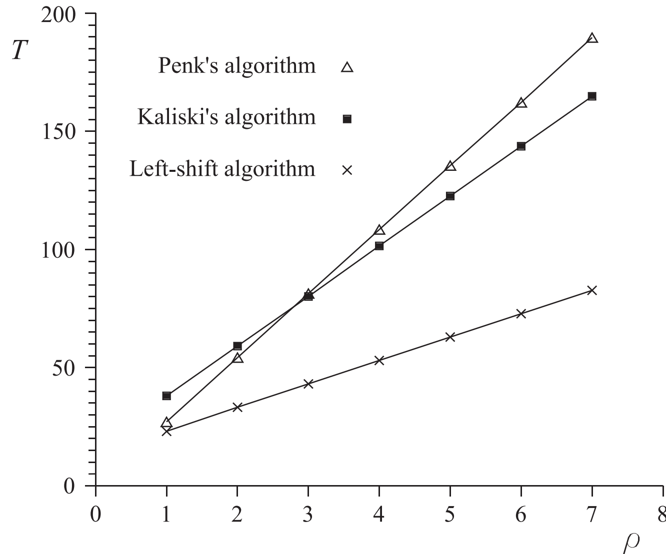

Figure 3: Average number of execution cycles T of the three algorithms as a function of the ratio ρ

algorithms, the modular inverse computation for individually algorithms is strongly dependent on addition/subtraction operations. That in such cases Left-shift algorithm is twice faster than Kaliski's algorithm and 2.7-times faster than Penk's algorithm. The statistical analysis of Left-shift algorithm for large integers of cryptographic was performed. The results of the analysis are presented in Table 2. The first column contains values of primes, the second column gives the word length and the last column gives number of inverses. Other columns have the same meaning as columns in Table 1. The average number of '+/−' operations grows with n approximately linearly. The multiplicative coefficient is ≈ 0.7 for all three primes. The average number of shifts is nearly equal to $2n$. Similar results hold for primes $p < 2^{14}$.

Table 2: Results of Left-shift algorithm for three cryptographic primes

Primes	n	$+/-$		*total shifts*		*inverses*
		min - max	*avg.*	*min - max*	*avg.*	
$2^{192} - 2^{64} - 1$	192	64 - 182	132.9	343 - 382	380	3,929,880
$2^{224} - 2^{96} + 1$	224	81 - 213	154.8	408 - 446	441	4,782,054
$2^{521} - 1$	521	18 - 472	387.5	999 - 1040	1029	4,311,179

3.3 HW Implementation

Left-shift algorithm is optimized in terms of reducing the number of additions and subtractions, which are critical in integer arithmetic due to carry propagation in long computer words. Other optimization criteria included making the evaluation of tests during the calculation as simple as possible and minimizing data dependencies to enable calculation in parallel calculations.

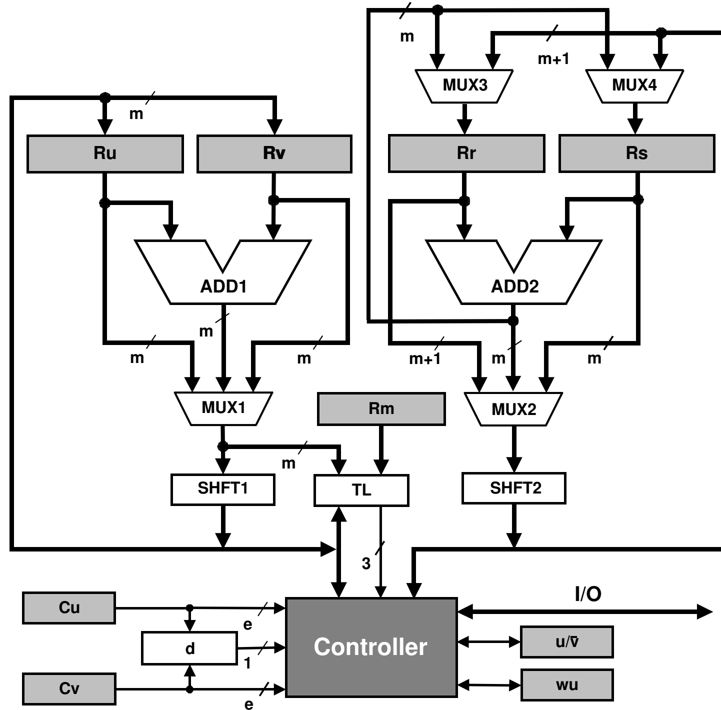

Figure 4: The circuit implementation of Left-shift algorithm

Figure 4 shows the circuit implementing the computation of classical modular inverse. Only data paths for computing the classical modular inverse according to Left-shift algorithm are shown. The system consists of three basic parts. The first two parts form the well-known "butterfly" [2, 3], typical for algorithms based on the EEA; the third part consists of the controller and support circuitry. "Master" half of the "butterfly" calculates the $\gcd(m, a)$ and consists of two m bit registers Ru, Rv, m bit adder/subtracter ADD1, multiplexer MUX1, and left-shift logic SHFT1. "Slave" half of the butterfly consists of $(m+1)$ bit register Rr and m bit register Rs, m bit adder/subtracter ADD2, multiplexers MUX2, MUX3, MUX4, and right/left-shift logic SHFT1. The controller controls the operation of the entire system. The controller part also includes an m bit mask register Rm with test logic provided test in step 3, two e bit counters Cu, Cv with the comparator d and single-bit flip-flops u/v̄ and wu.

3.4 The Mathematical Proof of the Left-shift Algorithm

The Left-shift algorithm has similar properties as the Euclidean Algorithm and algorithms derived from it, introduced in [5, 10, 19], where methods for their verification are also presented. By using similar proof techniques we have carried out a proof that Left-shift algorithm computes correctly the classical modular inverse.

For computing multiplicative inverse of an integer a in the finite field GF(p), where p is a prime, the following lemma is important.

Lemma 3.1 *If* $\gcd(p, a) = 1$ *and if* $1 = px + ab$, *then* $a^{-1} \bmod p = b \bmod p$.

The proof of the lemma is in [5]. By finding a pair of integers x and b that satisfy equations in Lemma 3.1, we prove that Left-shift algorithm computes classical inverse in GF(p). Individual iterations of Left-shift algorithm can be described by following system

(6) of recurrent equations and guarding conditions for quotients,

$$
\begin{array}{lllllll}
r_1 & = & p - aq_1 & 0 < r_1 < aq_1 & q_1 & = 2^{(\langle p\rangle - \langle a\rangle)} & q_1 > 1 \\
r_2 & = & |r_1| - aq_2 & 0 < r_2 < aq_2 & q_2 & = 2^{(\langle r_1\rangle - \langle a\rangle)} & q_2 > 1 \\
 & \vdots & & & & & \\
r_j & = & |r_{j-1}| - aq_j & 0 < |r_j| < a & q_j & = 2^{(\langle r_{j-1}\rangle - \langle a\rangle)} & q_j > 1 \\
\hline
r_{j+1} & = & a - |r_j|q_{j+1} & 0 < |r_{j+1}| < |r_j|q_{j+1} & q_{j+1} & = 2^{(\langle a\rangle - \langle r_j\rangle)} & q_{j+1} > 1 \\
r_{j+2} & = & |r_{j+1}| - |r_j|q_{j+2} & 0 < |r_{j+2}| < |r_j|q_{j+2} & q_{j+2} & = 2^{(\langle r_{j+1}\rangle - \langle r_j\rangle)} & q_{j+2} > 1 \\
 & \vdots & & & & & \\
r_k & = & |r_{k-1}| - |r_j|q_k & 0 < |r_k| < |r_j| & q_k & = 2^{(\langle r_{k-1}\rangle - \langle r_j\rangle)} & q_k > 1 \\
\hline
r_{k+1} & = & |r_j| - |r_k|q_{k+1} & 0 < |r_{k+1}| < |r_k|q_{k+1} & q_{k+1} & = 2^{(\langle r_j\rangle - \langle r_k\rangle)} & q_{k+1} > 1 \\
r_{k+2} & = & |r_{k+1}| - |r_k|q_{k+2} & 0 < |r_{k+2}| < |r_k|q_{k+2} & q_{k+2} & = 2^{(\langle r_{k+1}\rangle - \langle r_k\rangle)} & q_{k+2} > 1 \\
 & \vdots & & & & & \\
r_l & = & |r_{l-1}| - |r_k|q_l & 0 < |r_l| < |r_k| & q_l & = 2^{(\langle r_{l-1}\rangle - \langle r_k\rangle)} & q_l > 1 \\
\hline
 & \vdots & & & & & \\
\hline
 & \vdots & & & & & \\
r_m & = & \cdots & & & & \\
\hline
r_{m+1} & = & \cdots & & & & \\
 & \vdots & & & & & \\
r_n & = & |r_{n-1}| - |r_m| & 0 < |r_n| < |r_{n-1}| & q_n & = 2^{(\langle r_{n-1}\rangle - \langle r_m\rangle)} & q_n = 1 \\
\hline
r_{n+1} & = & |r_{n-1}| - |r_n|q_{n+1} & 0 < |r_{n+1}| < |r_n|q_{n+1} & q_{n+1} & = 2^{(\langle r_{n-1}\rangle - \langle r_n\rangle)} & q_{n+1} > 1 \\
r_{n+2} & = & |r_{n+1}| - |r_n|q_{n+2} & 0 < |r_{n+2}| < |r_n|q_{n+2} & q_{n+2} & = 2^{(\langle r_{n+1}\rangle - \langle r_n\rangle)} & q_{n+2} > 1 \\
 & \vdots & & & & & \\
r_o & = & |r_{o-1}| - |r_n|q_o & |r_o| = 1 & q_o & = 2^{(\langle r_{o-1}\rangle - \langle r_n\rangle)} & q_o > 1 \\
\hline
0 & = & |r_n| - |r_o|q_{o+1}, & & & &
\end{array}
\tag{6}
$$

where $r_1, r_2, \ldots, r_{o+1}$ are remainders, $q_1, q_2, \ldots, q_{o+1}$ are quotients, $\langle r_i\rangle$ is the number of bits needed for binary representation $|r_i|$. If the recursive definition of r_i is unrolled up to p and a, each r_i can be expressed by a Diophantine equation $r_i = pf_i + aq_i$, where $f_i = f_i(q_1, q_2, \ldots, q_i)$, and $g_i = g_i(q_1, q_2, \ldots, q_i)$.

The last non-zero remainder equal r_o fulfils the following theorem:

Theorem 3.1 $\gcd(p, a) = 1$ *iff* $|r_o| = 1$.

Proof

$$
\begin{array}{lll}
\gcd(p,a) & = & \gcd(r_1, a) = \gcd(r_2, a) = \cdots = \gcd(r_{j-1}, a) \\
 & = & \gcd(a, |r_j|) = \gcd(|r_{j+1}|, |r_j|) = \cdots = \gcd(|r_{k-1}|, |r_j|) \\
 & = & \gcd(|r_j|, |r_k|) = \gcd(|r_{k+1}|, |r_k|) = \cdots = \gcd(|r_{l-1}|, |r_k|) \\
 & = & \gcd(|r_k|, |r_l|) = \cdots \\
 & \vdots & \\
 & = & \cdots = \gcd(|r_{n-1}|, |r_m|) = \gcd(|r_n|, |r_m|) = \gcd(|r_{n-1}|, |r_n|) \\
 & = & \gcd(|r_{n-1}|, |r_n|) = \gcd(|r_{n+1}|, |r_n|) = \cdots = gcd(|r_{o-1}|, |r_n|) \\
 & = & \gcd(|r_n|, |r_o|) = \gcd(|r_o|, 0) \\
 & = & |r_o| = 1.
\end{array}
$$

□

The previous statement assumed the following trivial properties of gcd:

$$\begin{aligned} \gcd(0,d) &= |d| \text{ for } d \neq 0, \\ \gcd(c,d) &= \gcd(d,c), \\ \gcd(c,d) &= \gcd(|c|,|d|), \\ \gcd(c,d) &= \gcd(c+ed,d), \end{aligned}$$

where c, d, and e are integers. The description of the properties is introduced in [5, 19].

The fact that the guarding conditions for quotients q_i guarantee correct values of remainders r_i follows from the Lemma 3.2:

Lemma 3.2 *Let c and d be positive integers with binary representations $c = 2^i + c_{i-1}2^{i-1} + \cdots + c_0$ and $d = 2^j + d_{j-1}2^{j-1} + \cdots + d_0$. Assume $i \geq j$. Let $q = 2^{(i-j)}$ and $e = c - qd$. Then:*

$$|e| < qd \text{ and } |e| < c.$$

Proof Follows easily from identity

$$\sum_{k=1}^{i} \frac{1}{2^k} = 1 - \frac{1}{2^i},$$

which is proven for example in [19]. □

The computation specified in Equations (6) can be expressed in the form of Table 3, which gives the expressions for integer values f_i and g_i.

Table 3: The computation of f_o, g_o, and r_o

i	r_i	f_i	g_i
1	r_1	1	$-q_1$
2	r_2	± 1	$\pm q_1 - q_2$
3	r_3	± 1	$\pm q_1 \pm q_2 - q_3$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
j	r_j	± 1	$\pm q_1 \pm q_2 \pm \ldots - q_j$
$j+1$	r_j	$\pm q_{j+1}$	$1 \pm q_{j+1}(\pm q_1 \pm q_2 \pm \ldots - q_j)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
k	r_k	f_k	g_k
$\vdots$	$\vdots$	$\vdots$	$\vdots$
o	r_o	f_o	g_o
$o+1$	0	f_{o+1}	g_{o+1}.

Since $r_o = pf_o + ag_o$, it follows that:
if $(r_o = 1)$ then
  $x = f_o$, $b = g_o$, and $a^{-1} \bmod p = g_o \bmod p$,
if $(r_o = -1)$ then
  $x = -f_o$, $b = -g_o$, and $a^{-1} \bmod p = (-g_o) \bmod p$. ■

It is the last step of the proof that Left-shift algorithm computes classical modular inverse. Finally, we take note of the principal features of the Left-shift algorithm from the point of view of the presented proof. The algorithm employs two's complement code for additions or subtractions. Therefore, the test whether an addition or subtraction is to be performed

becomes a simple sign test. If the signs of both operands are equal, we subtract one from the other and in the opposite case we add both operands. Equations in (6) respect this rule when using absolute values of operands. According to Lemma 3.2, successive values r_i of remainders decrease in such a way that $p > a > |r_1| > |r_2| > \cdots |r_o|$.

The selection of operands which will be rewritten with a new value of the computed remainder is based on a simple test. The write is performed into the operand which needs more bits for its binary representation. This fact is respected in (6) by the value q_i. If $q_i = 1$, the write is into one of operands without respect on their absolute values. This case is demonstrated for remainder r_n. The conditions given by inequalities of Lemma 3.2 for r_n are fulfilled, too. Hence it holds $p > a > |r_1| > |r_2| > \cdots > |r_{n-1}| > |r_n| > |r_{n+1}| > \cdots > |r_o|$.

4 Conclusion

In the first part of the presented paper a modular system based on the residual number system for solves a set of linear equation s exactly was proposed. The method of calculation and its algorithms on the whole were solved with regard to their implementation for a hardware parallel SIMD architecture of the modular system. In the design their efficiency was stressed, which determined also the simplicity of the parallel architecture. Both rounding and truncation errors were excluded from the calculation.

In the second part a new algorithm (Left-shift algorithm) for classical modular inverse was presented and its HW implementation was proposed. A mathematical proof that the proposed algorithm really computes classical modular inverse was performed. Computation of the modular inverse using the new algorithm is always faster and in the case of long words at least twice faster than other algorithms currently in use. The principles of the presented algorithm are used in a modular system for solving systems of linear equations without rounding errors [14, 15].

References

[1] "ANSI/IEEE Standard 754-1985, Standard for Binary Floating Point Arithmetic", *http://grouper.ieee.org/groups/754/.*

[2] B. Bruner, A. Curiger, M. Hofstetter, "On Computing Multiplicative Inverse in GF(2^m)", *IEEE Trans. Computer*, vol. 42, pp. 1010–1015, 1993.

[3] J. D. Dworkin, P. M Glaser, M. J. Torla, A. Vadekar, R. J. Lambert, S. A. Vanstone, "Finite Field Inverse Circuit", *US Patent 6,009,450*, Dec. 28, 1999.

[4] I. Z. Emiris, V. Y. Pan, Y. Yu, "Modular Arithmetic for Linear Algebra Computations in the Real Field", *J. Symbolic Computation*, vol. 26, pp. 71–87, 1998.

[5] R. T. Gregory, E. V. Krishnamurthy, "Methods and Applications of Error-Free Computation", *Springer-Verlag, New York, Berlin, Heidelberg, Tokyo*, 1984.

[6] A. A. Gutub, A. F. Tenca, Ç. K. Koç, "Scalable VLSI Architecture for GF(p) Montgomery Modular Inverse Computation", *Proceeding of the IEEE Computer Society Annual Symposium on VLSI*, 2002.

[7] N. J. Higham: "Accuracy and Stability of Numerical Algorithms, *SIAM, PA*, 1996.

[8] J. A. Howell, R. T. Gregory, "An algorithm for solving linear algebraic equations using residue arithmetic I-II", *BIT*, vol. 9, no. 3, 4, pp. 200–224 and 324–337, 1969.

[9] B. S. Kaliski Jr., "The Montgomery Inverse and Its Application", *IEEE Transaction on Computers*, vol. 44, no. 8, pp. 1064–1065, 1995.

[10] D. E. Knuth, "The Art of Computer Programming **2** / Seminumerical Algorithms", *Addison-Wesley, Reading, Mass. Third edition*, 1998.

[11] R. Lórencz, "New Algorithm for the Classical Modular Inverse", *Proc. of Workshop on Cryptographic Hardware and Embedded Systems – CHES, August 13-15, 2002, Redwood Shores, CA, USA, Springer-Verlag 'LNCS, vol. 2523'*, pp. 57–70, 2002.

[12] R. Lórencz, "Architecture for Generating the Multiplicative Inverse over a Finite Field GF(p)", Patent No. 294898, 2005, Czech Republic.

[13] R. Lórencz, C. Reckleben, K. Hansen, "A novel Extraction Method for BJT-Parameters", *Journal of Electrical Engineering* , vol. 51, no. 1–2, pp. 21-29, 2000.

[14] R. Lórencz, M. Morháč, "Modular System for Solving Linear Equations Exactly, II. Hardware realization", *Computers and Artificial Intelligence*, vol. 11, no. 5, pp. 497–507, 1992.

[15] M. Morháč, R. Lórencz, "Modular System for Solving Linear Equations Exactly, I. Architecture and Numerical Algorithms", *Computers and Artificial Intelligence*, vol. 11, no. 4, pp. 351–361, 1992.

[16] M. Morháč, R. Lórencz, "Architecture of Processor Unit", *Architektúra procesorovej jednotky, Úřad průmyslového vlastnictví, 1987, Patent 257355.*

[17] J. C. Nash: "Compact Numerical Methods for Computers, Linear Algebra and Function Minimisation", *Adam Hilger, Bristol and New York*, 1990.

[18] M. Newman: "Solving exquations exactly", *National Bureau of Standarts, 71B*, pp. 171-179, 1967.

[19] K. H. Rosen, "Elementary Number Theory and Its Applications", *Addison-Wesley Publishing Company*, 1993.

[20] G. W. Stewart, J. Sun: "Matrix Perturbation Theory", *Academic Press, Boston*, 1990.

[21] G. Villard, "Exact parallel solution of linear systems", *In J. Della Dora and J. Fitch, editors, Computer Algebra and Parallelism, New York, Academic Press*, pp. 197–205, 1989.

[22] Z. Yan, D. Sarwate, "New Systolic Architectures for Inversion and Division in GF(2^m)", *IEEE Trans. on Computers*, vol. 5, no. 11, pp. 1514–1519, 2003.

[23] D. S. Watkins: "Fundamentals of Matrix Computations", *J. Willey, N.Y.*, 1991.

[24] Ç. K. Koç, A. Güvenç, B. Bakkaloğlu, "Exact Solution of Linear Equations on Distributed-Memory Multiprocessors", *Parallel Algorithms and Applications*, vol. 3, pp. 135–143, 1994.

Ing. Róbert Lórencz, CSc.

Date and place of birth: August 10, 1957 in Prešov, Slovak Republic

Affilation: Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Computer Science and Engineering

Postal address: Karlovo náměstí 13, 121 35 Praha 2, Czech Republic

E-mail address: lorencz@fel.cvut.cz

Education:

1981 – Ing. (M.S.) at Czech Technical University, Faculty of Electrical Engineering
1991 – CSc. (Ph.D.) at Institute of Measurement Science, SAS

Brief chronology of employment:

1982 – 1983	Research worker	Nuclear Center, Charles University
1983 – 1990	Research worker	Institute of Physics, SAS
1991 – 1995	System analyst	Research Center, Škoda Auto
1995 – 1998	Assistant professor	FEEI, Technical University of Košice
since 1998	Assistant professor	Department of Computer Science, CTU FEE

Stages and study leaves:

1985	Study stay	Collage on Microprocessors, ICTP Terst
1990	Study stay	Gesellschaft für Schwerionenforschung, Darmstadt
1990 – 1991	Guest researcher	Ludwig-Maximilians-Universität, München
1996 – 1998	Guest researcher	Deutsches Elektronen-Synchrotron, Hamburg

Teaching experience:

Lectures:	since 2001	Computer Architecture
	2001 – 2003	Special Architectures
	2001 – 2003	Design of Microcomputer Systems
	2002 – 2004	Machine Oriented Languages
	since 2002	Applied Numerical Mathematics
	since 2004	Machine Code and Data

Supervising of graduate students: 7 successful graduated (6 M.S., 1 B.S.) students
Supervising of Ph.D. students: currently 7 Ph.D. (5 supervisor, 2 supervisor specialist) students

Publication activity:

Over 55 journal and conference papers, 27 SCI citations, and 2 patents.

Main grant support:

Research in the Area of the Prospective Information and Navigation Technologies, since 2005.

Research activities:

Head of research group Applied Numerics and Cryptography

Research areas:

Numerical algorithms based on residual arithmetic for error-free computation and computer graphics algorithms. Methods of parameter extraction. Efficient algorithms for elementary arithmetic operations in GF(p) and GF(2^m) suitable for hardware design of accelerators used in cryptography and numerical computations.