Czech Technical University in Prague
Faculty of Information Technology
Department of Computer Systems

Properties and Implementation Aspects of Residue Arithmetic for a Hardware Solver of Systems of Linear Equations

by

Jiří Buček

A dissertation thesis submitted to
the Faculty of Information Technology, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

Dissertation degree study programme: Informatics

Prague, August 2017

Supervisor:

prof. Ing. Róbert Lórencz, CSc.
Department of Computer Systems
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Prague 6
Czech Republic

Copyright © 2017 Jiří Buček

Abstract

This dissertation thesis focuses on implementation aspect of hardware-based error-free solving of systems of linear equations. Error-free solution of linear systems is often needed in case of large, dense and ill-conditioned systems, where rounding errors can lead to long run times due to stability problems, or even hinder the solution completely. We explored the modular arithmetic approach using the Residue Number System (RNS).

We have analyzed and implemented several architectures for modular multiplication and modular inverse, which are needed to implement the elimination algorithm for solving the linear systems. We have redesigned the architecture of a residual processor for solving systems of linear congruences. This is a part of a modular system for solving systems of linear equations using the residue number system.

We have analyzed the implementation results in FPGA and ASIC platforms for different parameters such as word length or matrix dimension. The quality of hardware implemented algorithms was measured using the metrics of time and area and also the time-area product.

Our analysis is will serve as a base for improvement of the modular system for solving systems of linear equations. The resulting system architecture permits error-free solution of dense systems of linear equations of sizes of more than 1000 equations in reasonable configuration in a few seconds using contemporary technology.

Keywords:

modular arithmetic, error-free computation, linear algebra, system of linear equations, modular inverse, Montgomery inverse, Montgomery multiplication, FPGA, ASIC.

Acknowledgements

First of all, I would like to express my gratitude to my dissertation thesis supervisor, Dr. Róbert Lórencz. He has been a constant source of encouragement and insight during my research and helped me with numerous problems and professional advancements.

I would also like to thank to my collaborators for valuable contribution to my work. This research has also been partially supported by the Czech Technical University SGS grant No. SGS13/101/OHK3/1T/18 and by the Czech Science Foundation project no. P103/12/2377.

Finally, my greatest thanks go to my family members, for their infinite patience and care.

Contents

Abbreviations **xiii**

1 Introduction **1**
1.1 Structure of the Dissertation thesis 2

2 Theoretical Background and State-of-the-Art **3**
2.1 Systems of linear equations 3
2.2 Montgomery multiplication 7
2.2.1 The original algorithm 7
2.2.2 Choice of radix 8
2.2.3 Binary radix Montgomery multiplication 8
2.2.4 Final subtraction 9
2.2.5 Choice of encoding 9
2.3 Modular Inverse 10
2.3.1 Montgomery Inverse 13
2.3.2 Left-Shift Inverse 14

3 Previous Results and Related Work **15**
3.1 Modular system for solving systems of linear equations 15
3.2 Montgomery multiplication 16
3.2.1 Software approaches 16
3.2.2 Hardware approaches 17
3.3 Modular Inverse 20
3.3.1 Subtraction-Free Montgomery Inverse 20
3.3.2 Left-Shift Inverse 20

4 Overview of Our Approach **25**
4.1 FPGA implementation of a pipelined Montgomery multiplier 28
4.2 Montgomery multiplier with modified Carry-Save encoding 28

4.3 HW optimization of Left-Shift Inverse 30
4.3.1 Datapath 32
4.3.2 Controller 32
4.4 Comparing Montgomery inverse architectures 34
4.5 Solver of systems of linear equations 35

5 Author's Relevant Papers **39**
5.1 RP1 – Montgomery Multiplication on FPGA with Modified Carry-Save Encoding 41
5.2 RP2 – Comparing Subtraction-Free and Traditional AMI 45
5.3 RP3 – Dedicated Hardware Implementation of a Linear Congruence Solver in FPGA 49
5.4 RP4 – Comparison of FPGA and ASIC Implementation of a Linear Congruence Solver 54
5.5 RP5 – An ASIC Linear Congruence Solver Synthesized with Three Cell Libraries 59
5.6 RP6 – Design of a Residue Number System Based Linear System Solver in Hardware 64

6 Conclusions **79**
6.1 Summary 79
6.2 Contributions of the Dissertation Thesis 81
6.3 Future Work 81

Bibliography **83**

Reviewed Publications of the Author Relevant to the Thesis **87**

Remaining Publications of the Author Relevant to the Thesis **89**

Remaining Publications of the Author **91**

List of Figures

3.1 Modular system for solving systems of linear equations exactly [1, 2]. 16
3.2 Scalable pipelined implementation of the Montgomery multiplier. 18
3.3 Pipeline processing element. 19

4.1 Internal structure of adders: a) conventional carry-save adder, b) modified carry-save adder. 29
4.2 Adder tree using a) carry-save encoding, b) modified carry-save encoding. . . . 29
4.3 Multiplier architecture with modified CSA adder tree. 30
4.4 LSI datapath schematic 33
4.5 Master part of AMI: a) Two subtractors, b) one subtractor with swappable inputs, c) subtraction-free. 34
4.6 Arhitecture of the residual processor 36

5.1 Relationships among author's relevant papers. 40

List of Tables

2.1 Computational complexity of individual steps of solving a system of linear equations using RNS 6

3.1 Left-Shift Inverse operation summary 21

4.1 Left-Shift Inverse HW optimized operation summary 32

6.1 Load, elimination, and read times for the MS architecture with multiple RPs. The Total architecture is the sum of each row for the case of 1 RP. 80

6.2 FPGA (Xilinx xc6vlx240t) area occupation of MS architecture with multiple RPs. The last column is only the RP without the rest of the MS. BRAM is the number of block RAM primitives computed as the number of RAMB18E1 plus 2 × RAMB36E1. 80

List of Algorithms

2.1 Gauss-Jordan elimination algorithm modulo m without row swapping. . . 5
2.2 Montgomery reduction 8
2.3 High-radix Montgomery multiplication 9
2.4 Binary-radix Montgomery multiplication 9
2.5 Extended Euclidean algorithm for modular inverse 11
2.6 Extended Binary GCD Euclidean Algorithm 12
2.7 Montgomery modular inverse 13
3.1 Rutishauser modification of Gauss-Jordan elimination modulo m. 16
3.2 Multi-word Montgomery multiplication 18
3.3 Binary Montgomery multiplication with carry-save encoding 19
3.4 Subtraction-Free AMI 21
3.5 Left-Shift Inverse algorithm 22
4.1 Left-shifting inversion algorithm, HW optimized variant 31

Abbreviations

Number Sets

$\mathbb{N}$	Natural numbers set
$\mathbb{N}_0$	Natural numbers set $\cup \{0\}$
$\mathbb{Z}$	Integer numbers set
$\mathbb{Z}_m$	Least nonzero residue number set with a module of m
$\mathbb{S}_m$	Symmetric residue number set with a module of m
$\mathbb{Q}$	Rational numbers set
$\mathbb{R}$	Real numbers set

Common Mathematical Functions and Operators

10_2	Numbers' radices are designated with a subscript
$\|x\|$	Absolute value of x
$\|x\|_m$	Least nonnegative residue of x modulo m, i.e. $x \bmod m$
$\boldsymbol{b}$	Vector $\boldsymbol{b}$
b_i	the ith element of vector $\mathbf{b}$
$\mathbf{A}$	Matrix $\mathbf{A}$
$a_{i,\,j}$	Element of matrix $\mathbf{A}$ at the ith row, and the jth column
$\mathbf{A}^{-1}$	Inverse matrix to matrix $\mathbf{A}$
$\mathbf{A}^T$	Transposed matrix to matrix $\mathbf{A}$
$\max\{a,\, b\}$	Maximum of a and b, a when $a \geq b$, b when $a < b$
$\min\{a,\, b\}$	Minimum of a and b, a when $a \leq b$, b when $a > b$
$O(x)$	The big O notation
$\Theta(x)$	The big Θ notation

Mathematical Terminology

M	A product of individual moduli $M = \prod_{i=1}^{Q} m_i$

Miscellaneous Abbreviations

AMI	Almost Montgomery Inverse
ASIC	Application-Specific Integrated Circuit
AU	Arithmetic Unit
CCU	Central Control Unit
CU	Communication Unit
CSA	Carry Save Adder
DSP	Digital Signal Processor
EEA	Extended Euclidean Algorithm
FP	Floating Point
FPGA	Field-Programmable Gate Array
FPU	Floating Point Unit
FSM	Finite State Machine
GCD	Greatest Common Divisor
GF	Galois Field, a finite field
GJ	Gauss-Jordan
HDL	Hardware Description Language
HS	Host System
HW	Hardware
LSI	Left-Shift Inverse
MI	Modular Inverse
MM	Montgomery Multiplication
MMI	Montgomery Modular Inverse
MRC	Mixed Radix Conversion
MRS	Mixed Radix System
MS	Modular System
NTL	Number Theoretic Library
RNS	Residue Number System
RAM	Random Access Memory, a read-write memory
RP	Residal Processor
RSA	Rivest-Shamir-Adleman cryptosystem
RTL	Register Transfer Level
SLC	System of Linear Congruences
SLE	System of Linear Equations
TSMC	Taiwan Semiconductor Manufacturing Company
VHDL	Very High Speed Integrated Circuits Hardware Description Language

XST Xilinx Synthesis Technology, a HDL synthesis tool
DDR Double Data Rate, a type of memory interface

Chapter 1

Introduction

In the field of scientific computation, it is always needed to consider the influence of round-off errors on the solved problems. Solving systems of linear equations is one of the most frequent problems in scientific computation. Traditionally, solving of such systems is done in floating point arithmetic which brings its associated rounding errors.

The input data for solving can arise from measurements, model simulation parameters, or other sources. It is encountered in many different fields such as mechanical and electrical modeling and simulation, modeling of plasma behavior in astrophysics [3].

Although there are algorithms that minimize the impact of rounding errors on the error of the solution, in some cases this is not enough. Error-free solution of linear systems is often needed in case of large, dense and ill-conditioned systems, where rounding errors can lead to long run times due to stability problems, or even hinder the solution completely. One of the methods to perform error-free computation is using the Residue Number System (RNS).

When designing hardware architectures for computing in RNS, we have to implement basic operations in modular arithmetic. In this work, we focus on modular multiplication and modular inverse. Subsequently, these operations are used to implement an architecture for solving systems of linear congruences with prime modulus, which serves as the basis for solving systems of linear equations using RNS.

For modular multiplication, the Montgomery multiplication algorithm [4] is often used. As opposed to the conventional modular multiplication, the Montgomery's algorithm does not use division by the modulus N, rather it uses division by 2^i, which is faster, since it is done merely by shifting (i is an integer). The reduction step is performed depending on the least significant digit of the intermediate result, rather than on the most significant one, as in the case of a conventional multiply. This causes another speedup, since the critical path does not necessarily go through the full carry chain of the adder.

Calculation of the modular inverse is needed for solving systems of linear congruences as well as conversion from RNS to integer numbers. It is also used in cryptography especially in computing point operations on elliptic curves [5], or in accelerating the modular exponentiation operation using the so-called addition-subtraction chains [6].

The quality of hardware implemented algorithms is measured using the metrics of time, area and power consumption. The underlying technology is often an Application Specific Integrated Circuit (ASIC) or a reconfigurable architecture such as a Field Programmable Gate Array (FPGA). Other technology includes general purpose processors, DSPs or embedded microcontrollers.

1.1 Structure of the Dissertation thesis

The thesis is organized as follows. Chapter 2 introduces basic definitions and terminology regarding Montgomery multiplication, modular inverse and systems of linear equations. Chapter 3 summarizes the previous work in this field. Chapter 4 presents our architecture modifications. Chapter 5 presents our results in the form of a collection of published papers, and Chapter 6 concludes with outlines for future work.

Chapter 2

Theoretical Background and State-of-the-Art

In this chapter, we summarize the basics covering the problem of solving systems of linear equations in residue (modular) arithmetic. We also visit some of the individual operations needed, namely modular multiplication and modular inverse.

2.1 Systems of linear equations

Let us assume the following linear system with a square matrix of coefficients $\mathbf{A}$ and a vector of right-hand sides $\boldsymbol{b} \neq \mathbf{0}$. The task is to compute the solution vector $\boldsymbol{x}$.

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b} \tag{2.1}$$

Most often the input values, i.e. the coefficients and right-hand sides of individual linear equations, are floating point numbers, sometimes integers. (We are not considering complex or irrational numbers.) Traditionally, solving of such systems is also done in floating point arithmetic, which brings its associated rounding errors.

There are many methods to solve such systems – direct methods like the Cramer's rule or Gaussian elimination, matrix decomposition methods like the LU decomposition (essentially equivalent to Gaussian elimination), QR decomposition or singular value decomposition, and also iterative methods like Jacobi method, Gauss-Seidel method, or conjugate gradient method.

Although there are algorithms that minimize the impact of rounding errors on the error of the solution, in some cases this is not enough. Error-free solution of linear systems is often needed in case of large, dense and ill-conditioned systems, where rounding errors can lead to long run times due to stability problems, or even hinder the solution completely. One of the methods to perform error-free computation is using the residue number system (RNS), and thus using modular arithmetic.

When considering which methods and algorithms are suitable for RNS, we must know what operations are required, and how easily can they be implemented in RNS. The operations needed for iterative methods include minimization of a vector of residuals that must fullfill a termination criterion, usually a small absolute value. Magnitude comparison operations are quite expensive in RNS, and the result is inherently imprecise, which is against the effort to get error-free solution. The QR and SVD decomposition methods need the square root operation, which cannot be implemented easily in RNS.

Matrix $\mathbf{A}$ and vector $\boldsymbol{b}$ can be given as rational or floating point numbers. In any case, the system (2.1) it can be transformed to an equivalent linear system with integer numbers by scaling individual rows of $\mathbf{A}$ together with corresponding elements of $\boldsymbol{b}$. Therefore we can safely assume that all numbers are integers. Nevertheless, the elements of solution vector $\boldsymbol{x}$ will be rational in general.

First, we will convert the input system into RNS, then we will use modular arithmetic, to compute the solution in RNS, and finally we will convert the solution from RNS to rational numbers. In order to get correct result in full precision, we need a sufficiently large modulus M, whose value can be derived from the Hadamard's inequality for the determinant D of matrix $\mathbf{A}$:

$$D^2 \leq \prod_{i=1}^{n} \sum_{k=1}^{n} a_{ik}^2 \tag{2.2}$$

The largest value M, that can be encountered in the computation of the linear system, is bound [7, 1] by

$$M > 2 \max \left\{ \begin{array}{ll} n^{\frac{n}{2}} \max(a_{ij})^n, & i,j = 1,2,...,n, \\ n(n-1)^{\frac{n-1}{2}} \max(a_{ij})^{n-1} \max(b_i), & i,j = 1,2,...,n \end{array} \right\} \tag{2.3}$$

and

$$\gcd(M, D) = 1. \tag{2.4}$$

The computation will be done using multiple-modulus residue arithmetic with the moduli forming a vector $\boldsymbol{\beta} = (m_1, m_2, ..., m_r)$, where m_i are distinct primes and $\prod_i m_i = M$. In practice, it would be unnecessary to enforce that the determinant D be coprime to M rigorously. If $\gcd(M, D) \neq 1$, it means that for some m_s, $\gcd(m_s, D) \neq 1$. We can then exclude this m_s form our moduli vector $\boldsymbol{\beta}$ and continue with the remaining moduli.

In this work, we will use a variant of Gaussian elimination – the Gauss-Jordan elimination algorithm. Gaussian elimination converts a matrix to its row echelon form, whereas Gauss-Jordan elimination converts it to a unity matrix (provided the matrix was square and full rank, i.e. invertible). If applied to the augmented matrix $A = (\mathbf{A}|\boldsymbol{b})$ of the linear system (2.1), it produces the solution vector $\boldsymbol{x}$ of the system.

The process of solving (2.1) using multi-modulus residue arithmetic can be done using the following method.

Algorithm 2.1 Gauss-Jordan elimination algorithm modulo m without row swapping.

Inputs: A is the augmented matrix, n is matrix dimension.
Output: Solution vector $\boldsymbol{x}$ is the last column in A.
1. assuming pivot row j is found during loading of matrix
2. **for** $k = 1..n$ **begin**
3. $A(j) = |A_{j,k}^{-1}\ A(j)|_m$
4. **for** $i = 1..n$ except j **begin**
5. $A(i) = |A(i) - A_{i,k}\ A(j)|_m$
6. test new pivot, set pivot row j_{new}
7. **end**
8. $j = j_{\text{new}}$
9. **end**
10. **return** $\boldsymbol{x}$ **as** $x_i = A_{i,n+1}$ **for** $i = 1..n$

1. Input scaling – conversion of the input linear system from floating point (rational) numbers to integers.
2. Input conversion – conversion of the system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ to systems of congruences $\mathbf{A}\boldsymbol{x_k} \equiv \boldsymbol{b} \pmod{m_k}$
3. Solving SLC – solving r systems of linear congruences, computing $\boldsymbol{x}_k$ and $D_k = |\det A|_{m_k}$, and also the product $\boldsymbol{z}_k = |D_k\boldsymbol{x}_k|_{m_k}$.
4. Output conversion – conversion of $|D_k|_{m_k}$ and $\boldsymbol{z}_k = |D_k\boldsymbol{x}_k|_{m_k}$ into integers D and $\boldsymbol{z}$
5. Solution finalization – conversion of D and $\boldsymbol{z}$ into the final solution vector $\boldsymbol{x}$

Input scaling is done by multiplying the coefficients the right hand side of each equation with their common denominator (or a multiple thereof) in order to get integers. This can be done for any rational coefficients and, provided floating point inputs, can be done just by binary shifting. A sequential implementation requires $\mathrm{O}(n^2)$ multiplications.

Input conversion means computing the residui of all elements of $\mathbf{A}$ and $\boldsymbol{b}$ modulo individual moduli m_k. This involves computing the remainder from the division of each coefficient by m_i and requires $\mathrm{O}(rn^2)$ operations.

Solving systems of linear congruences mod m_k by Gauss-Jordan elimination requires $\mathrm{O}(rn^3)$ operations. During the computation in each modulus, the operations needed are modular multiplicative inverse for each of the n pivots, and modular multiplication, addition, and subtraction (or additive inverse). Additionally, either row swapping or result reordering is needed for the cases when a pivot is not found directly at the diagonal.

Output conversion involves reconstruction of the rational solution vector $\boldsymbol{x}$ from the individual solution vectors in each modulus. The values $|D|_{m_k}$ and $\boldsymbol{z}_k = |D\boldsymbol{x}|_{m_k}$ are converted into integers. A sequential Garner algorithm for Multi-Radix Conversion takes $\mathrm{O}(r^2n)$ operations.

Solution finalization results in the solution vector $\boldsymbol{x} = \frac{\boldsymbol{z}}{D}$. It is computed either as a precise rational number, or as a floating point number.

The computational complexity of the individual steps are summarized in table 2.1.

Step	1	2	3	4	5
Sequential computation	$O(n^2)$	$O(rn^2)$	$O(rn^3)$	$O(r^2n)$	$O(n)$
Row-parallel, modulo parallel architecture	$O(n^2)$	$O(n)$	$O(n^2)$	$O(rn)$	$O(n)$

Table 2.1: Computational complexity of individual steps of solving a system of linear equations using RNS

The operations needed are modular addition, subtraction, multiplication and multiplicative inverse. The time complexity of individual operations depends of the hardware architecture of each functional unit. We will consider only arithmetic of the Galois field GF(p), where p is a prime number. Assume the bit length of p being e bits, i.e. $e = \lceil \log_2 p \rceil$. Modular addition (and subtraction) is a combination of binary addition and a modulo reduction. The time and area complexity of the operation depends on the architecture. Many different architectures exist for binary adders (ripple-carry, carry look-ahead, carry-select and many others). Depending on the word size and other constraints, number of concurrent additions and other parameters, bit-serial adders can be appropriate.

Modular multiplication can be performed by a variety of methods, most of which are based on a combination of integer multiplication and modulo reduction. The choice of multiplication method depends of the operand length and performance requirements. Apart from schoolbook (long) multiplication, other methods exist mainly for large numbers, such as Karatsuba [8], Toom-Cook [9], and Schönhage-Strassen [10].

Modulo reduction means computing a remainder after division. This can be computed after multiplication, or interleaved during the product computation. For relatively small operands that fit in the machine word (tens of bits), a variant of the schoolbook multiplication is suitable, as it is easily interleaved with reduction steps to maintain the intermediate result within the machine word. Multiplication takes e additions, shifts and modular reductions in the computation.

The most complex elementary operation is the modular inverse. Several different algorithms exist for the inverse. In GF(p), algorithms based on the Extended Euclidean Algorithm are prevalent. Their operational complexity is derived from the binary GCD computation, whose average number of cycles is $1.4e$.

In the following sections, we will present a brief overview of the mathematical background of arithmetic operations that are suitable for a solver of linear equations in modular arithmetic. In section 2.2, we deal with Montgomery multiplication, a special type of modular multiplication that is often used due to its efficient hardware implementation. In section 2.3, we will present a summary of the basic principles of modular inverse.

2.2 Montgomery multiplication

Montgomery multiplication is a method for computing modular multiplication. It operates on a special representation of residue classes, often called the Montgomery domain. The Montgomery domain is a set of images, also called N-residues.

Definition 2.2.1. Let N, a be integers, $N > 1$, $0 \leq a < N$. Let R be an integer coprime to N and $R > N$. The N-residue (Montgomery image) $\overline{a}$ of a is defined as $\overline{a} = |aR|_N$.

Let R^{-1} and N' be integers satisfying

$$0 < R^{-1} < N, 0 < N' < R, RR^{-1} - NN' = 1 \qquad (2.5)$$

The Montgomery multiplication computes:

$$\overline{c} = \mathrm{MM}(\overline{a}, \overline{b}) = |\overline{a}\overline{b}R^{-1}|_N$$

It can be shown that $\overline{c}$ is the Montgomery image of $c = ab$, since

$$\begin{aligned}
\overline{c} &= |\overline{a}\overline{b}R^{-1}|_N \\
\overline{c} &= |aRbRR^{-1}|_N \\
\overline{c} &= |abR|_N \\
\overline{c} &= |cR|_N
\end{aligned}$$

Further we can see that the addition algorithm is unmodified, as $\overline{s} = |\overline{a} + \overline{b}|_N = |aR + bR|_N = |(a + b)R|_N = |sR|_N$ and thus we get the image of the sum by adding the images together. The same holds for subtraction.

In order to process integer numbers modulo N, we have to convert them to N-residues. After finishing computation in the Montgomery domain, we have to convert them back to the integer domain. We can perform forward as well as backward conversions using the same algorithm that we use for Montgomery multiplication. Observe that $\overline{a} = |aR|_N = \mathrm{MM}(a, R^2)$ and $a = |\overline{a}R^{-1}|_N = \mathrm{MM}(\overline{a}, 1)$. The constant $|R^2|_N$ can be precomputed and stored in the system as long as the modulus does not change too often.

Note: In this paper, we will no further distinguish between an integer a and its N-residue by its notation, since they are both integers of the same range. Their meaning is generally obvious, possible exceptions will be denoted explicitly.

2.2.1 The original algorithm

Montgomery proposed the algorithm for modular multiplication without trial division in his paper [4]. We choose R in such a way that operations modulo R are inexpensive;

possibly the machine word size or a power thereof. Again, R^{-1} and N' are determined by $0 < R^{-1} < N, 0 < N' < R, RR^{-1} - NN' = 1$. Montgomery has shown that if $0 \leq T < RN$, we can quickly compute $|TR^{-1}|_N$ using the following algorithm.

Algorithm 2.2 Montgomery reduction

function REDC(T)
1. $m := ||T|_R N'|_R$
2. $t := (T + mN)/R$
3. **if** $t \geq N$ **then return** $t - N$ **else return** t

In order to get the Montgomery product $\overline{c} = \mathrm{MM}(\overline{a}, \overline{b})$, we first perform an integer multiplication $T = ab$. Then we use Algorithm 2.2, $c = \mathrm{REDC}(T)$, to reduce T to $|TR^{-1}|_N$. Since $0 \leq a < N, 0 \leq b < N$, it holds that $0 \leq T < RN$ and thus the input condition is satisfied.

Notice that the reduction phase is presented independently of the multiplication phase. However, in this case we need sufficient space to store the result of the multiplication before the reduction is done. In order to save resources, we often interleave the two phases. The interleaving can be performed in a variety of means, some of them are presented below.

2.2.2 Choice of radix

As we stated above, we choose R to be such a value that operations modulo R are inexpensive to process. This usually means that R is 2^k, where k is the number of bits in the binary representation of the operands and the result. We perform operations modulo R in the step (1) of Algorithm 2.2. If we interleave the multiplication and reduction phases, we have a wide variety of options how to organize the computation.

Let r, e, w be positive integers, $r > 1$. If we choose $R = re$, then r is the radix of our arithmetic operations. Usually we set $r = 2^w$, $R = r^e = 2^{ew}$. We express the operands and product as $A = \sum_{i=0}^{e-1} a_i r^i$, $B = \sum_{i=0}^{e-1} b_i r^i$ and $S = \sum_{i=0}^{e-1} s_i r^i$. We can also express the modulus N as $N = \sum_{i=0}^{e-1} n_i r^i$ and take advantage of the digit representation by using only the least significant digit of N' for the reduction: $n'_0 = |-n_0^{-1}|_r$. The computation can be then expressed as follows: (Note that we do not perform the final subtraction, see below.)

2.2.3 Binary radix Montgomery multiplication

Binary Montgomery multiplication is a special case where $r = 2$. Assuming that $A = \sum_{i=0}^{e-1} a_i 2^i$, $B = \sum_{i=0}^{e-1} b_i 2^i$ and $S = \sum_{i=0}^{e-1} s_i 2^i$, we may describe the basic binary Montgomery multiplication algorithm as presented in Algorithm 2.4. The binary version has the advantage that we need not compute N', since the reduction is performed such that the quotient q is a single bit. Given $r = 2$ the multiplication factor n'_0 in step 3 is always $n'_0 = |-N^{-1}|_2 = 1$.

Algorithm 2.3 High-radix Montgomery multiplication

Inputs: A, B, N, e
Output: $S = \mathrm{MM}'(A, B)$
1. $S := 0$
2. **for** $i = 0$ **to** $e - 1$
3. $\quad q := |(s_0 + a_i b_0)n'_0|_r$
4. $\quad S := S + a_i B + qN$
5. $\quad S := S/r$
6. **end for**
7. **return** S

Algorithm 2.4 Binary-radix Montgomery multiplication

Inputs: A, B, N, k
Output: $S = \mathrm{MM}'(A, B)$
1. $S := 0$
2. **for** $i = 0$ **to** $k - 1$
3. $\quad q := s_0 + a_i b_0$
4. $\quad S := S + a_i B + qN$
5. $\quad S := S/2$
6. **end for**
7. **return** S

2.2.4 Final subtraction

Step (3) of Algorithm 2.2 guarantees that the result is always less than N. However, the comparison made here implies a subtraction with a consequent sign check of possibly very long operands. This is unfortunate, since this operation is very slow. There have been efforts to avoid the final subtraction step at all. The paper in [11] shows that for Algorithm 2.3, the result of the Montgomery multiplication is bound, $S = \mathrm{MM}'(A, B) < 2N$, as long as $A < 2N$, $B < 2N$ and $2N < r^{e-1}$. Another paper [12] improves this bound to $N < 2^{(e-1)w}$ at the cost that $w \geq 2$. This means that we have to relax the constraints on the operands A, B, while at the same time increasing the number of digits we compute. In both cases, we have to compute at least one more digit as opposed to the algorithm with final subtraction. It is important to know that we get properly reduced output once we convert the result S using $\mathrm{MM}'(S, 1)$. This is proved in [11].

2.2.5 Choice of encoding

We have multiple choices how to encode the operands and the (partial) product of the multiplication.

- Binary encoding

- Carry-save encoding
- Signed-digit encoding

We can also specify different encoding for individual variables in the algorithm. We decide how to encode each of the following:

- Multiplication operands
- Partial product

Binary encoding is non-redundant, having the advantage that we do not need additional conversions of the operands and/or the product. The drawback is that long binary addition can be slow because of the carry path in the adders. This can be mitigated by using faster adders such as carry look-ahead or carry completion adders. The area overhead and limited scalability of such adders however makes them unsuitable in most cases.

There are platforms, such as Field Programmable Gate Arrays (FPGAs), which have fast dedicated resources for carry chains and thus are (to a certain limit) less sensible to word length when performing binary addition.

Carry-save and signed-digit encodings are redundant, making conversions necessary when the result is to be used outside of the algorithm. Redundant representations are used to achieve higher working frequency of the resulting hardware since they avoid direct carry propagation through very long adders.

In order to implement more complex computation, such as a modular exponentiation, it is necessary to reuse the result of one multiplication in subsequent multiplications. Hence, it is desirable to avoid unnecessary conversions between redundant and non redundant forms of any of the operands. This can be solved by leaving both operands A, B as well as the result S in carry-save form. This approach ensures high throughput of modular exponentiation because the intermediate results can be directly used as operands of the next multiplication.

2.3 Modular Inverse

Modular (multiplicative) inverse $\mathrm{MI}(a) = |a^{-1}|_m$ of an integer a modulo m is defined as a number satisfying $aa^{-1} \equiv 1 \pmod{m}$, or $|aa^{-1}|_m = 1$. There are several algorithms for computing the modular inverse, mainly using Fermat's little theorem, or a variant of the Euclidean algorithm. In this work, we consider several different variants of the Euclidean algorithm. The most straightforward algorithm is the extended Euclidean algorithm (EEA), which uses repeated integer division to compute the greatest common divisor $\gcd(a, m)$ and two coefficients x, y of Bézout's identity

$$\gcd(a, m) = ax + my. \tag{2.6}$$

If $\gcd(a, m) = 1$, then it x is the multiplicative inverse of a modulo m. The coefficient y is usually not needed, therefore it is not computed at all. A reduced form of EEA, which computes only the inverse, is given in Algorithm 2.5.

Algorithm 2.5 Extended Euclidean algorithm for modular inverse

Input: Integers $a \in [1, m-1]$ and $m > 2$
Output: $r = |a^{-1}|_m$ or error
1. $u := m, v := a, r := 0, s := 1, k := 0$
2. **while** $(v \neq 0)$
3. $\quad q := \lfloor u/v \rfloor$
4. $\quad v_{\text{new}} := u - qv,\ u := v,\ v := v_{\text{new}}$
5. $\quad s_{\text{new}} := r - qs,\ r := s,\ s := s_{\text{new}}$
6. **if** $(u \neq 1)$ **return** "Not relatively prime"
7. **if** $(r < 0)$ $r := r + m$
8. **return** r

Notice the algorithm can be split in two parts – the *master* part computes the greatest common divisor, represented by the variables u and v in Algorithm 2.5. The *slave* part computes the modular inverse (variables r, s) and is controlled by the master part by the quotient q computed in step 3.

In hardware implementations, the division operation used in step 3 of Algorithm 2.5 would be unnecessary expensive. However, division by two is very cheap and is done by shifting. A binary variant of EEA can be formulated in Algorithm 2.6. This algorithm, attributed to M. Penk, is mentioned (in a slightly different form) in [13] (answer to exercise 39 of section 4.5.2). This algorithm performs halving in both the master part (variables a, b) and the slave part (f_1, f_2, g_1, g_2). In order to enable division by two, the algorithm tests if the variable is even. If not, a suitable odd value is added or subtracted to ensure the value can be halved. This can be simplified using the Montgomery inverse described in the next section.

Algorithm 2.6 Extended Binary GCD Euclidean Algorithm

Input: $a, b \in \mathbb{Z}$ and $a > b > 0$
Output: $\gcd(a, b) = xa + yb$ and $x, y \in \mathbb{Z}$
1. $k := 0, f_1 := 1, f_2 := 0, g_1 := 0, g_2 := 1$
2. **while** (a even $\wedge$ b even)
3. $\quad a := a/2$, $b := b/2$,$k := k + 1$
4. **while** ($a > 0$)
5. $\quad$ **if** (a even) **then**
6. $\quad\quad a := a/2$
7. $\quad\quad$ **if** (f_1 even $\wedge$ g_1 even) **then**
8. $\quad\quad\quad f_1 := f_1/2, g_1 := g_1/2$
9. $\quad\quad$ **else**
10. $\quad\quad\quad f_1 := (f_1 + b)/2, g_1 := (g_1 - a)/2$
11. $\quad$ **else if** (b even) **then**
12. $\quad\quad b := b/2$
13. $\quad\quad$ **if** (f_2 even $\wedge$ g_2 even) **then**
14. $\quad\quad\quad f_2 := f_2/2, g_2 := g_2/2$
15. $\quad\quad$ **else**
16. $\quad\quad\quad f_2 := (f_2 + b)/2, g_2 := (g_2 - a)/2$
17. $\quad$ **else**
18. $\quad\quad c := (a - b), d := f_1 - f_2, e := g_1 - g_2$
19. $\quad\quad$ **if** ($c \geq 0$) **then**
20. $\quad\quad\quad a := c, f_1 := d, g_1 := e$
21. $\quad\quad$ **else**
22. $\quad\quad\quad b := -c, f_2 := -d, g_2 := -e$
23. **return** $\gcd(a, b) := 2^k b$ and $x := f_2, y := g_2$

2.3.1 Montgomery Inverse

Montgomery modular inverse was introduced in 1995 by Kaliski [14] in the form $|a^{-1}2^n|_p$. The algorithm consists of two phases. The first phase computes $|a^{-1}2^k|_p$, where $k \geq n$. The second phase then divides this intermediate result by 2^{k-n} modulo p, yielding $|a^{-1}2^n|_p$. This algorithm can be used to compute modular inverse of an argument in the Montgomery domain while obtaining the resulting inverse in the integer domain: $|(x2^n)^{-1}2^n|_p = |x^{-1}|_p$. This is not always useful when we want to stay with the operands in the Montgomery domain.

Savaş and Koç presented a modified algorithm [15] (Algorithm 2.7) that maintains the Montgomery domain. This means that the modified Montgomery inverse takes the argument in the Montgomery domain and provides the result also in Montgomery domain. The first phase remains the same computing $|a^{-1}2^k|_p$, but the second phase is modified to multiply the intermediate result by 2^{2n-k} modulo p, yielding $\mathrm{MMI}(a) = |a^{-1}2^{2n}|_p$. When computing the inverse of an argument in the Montgomery domain, we get $|(x2^n)^{-1}2^{2n}|_p = |x^{-1}2^n|_p$.

The first phase of Motgomery inverse is referred to as Almost montgomery inverse $\mathrm{AMI}(a) = |a^{-1}2^k|_p$. Therefore $\mathrm{MMI}(a) = |\mathrm{AMI}(a)2^{2n-k}|_p = |\mathrm{MI}(a)2^{2n}|_p$.

Algorithm 2.7 Montgomery modular inverse

Input: $a \in [1, p-1]$ and $p > 2$ is prime, n number of bits in p
Output: $r \in [1, p-1]$, where $r = |a^{-1}2^{2n}|_p$

Phase I (Almost montgomery inverse)
1. $u := p, v := a, r := 0, s := 1, k := 0$
2. **while** $(v > 0)$
3. **if** (u **even**) **then** $u := u/2, s := 2s$
4. **else if** (v **even**) **then** $v := v/2, r := 2r$
5. **else if** $(u > v)$ **then** $u := (u-v)/2, r := r+s, s := 2s$
6. **else** $v := (v-u)/2, s := r+s, r := 2r$
7. $k := k+1$
8. **if** $(u \neq 1)$ **then return** "Not relatively prime"
9. **if** $(r \geq p)$ **then** $r := r - p$

Phase II (Savaş, Koç)
10. **while** $(k \neq 2n)$
11. $r := 2r$
12. **if** $(r \geq p)$ **then** $r := r - p$
14. $k := k+1$
15. **return** r

2.3.2 Left-Shift Inverse

In the previous algorithms, the master part, computing the greatest common divisor, was shifted to the right. On the contrary, the algorithm by Lórencz [16] shifts the master variables u and v to the left. The algorithm is referred to as Left-Shift Inverse (LSI) and will be described in section 3.3.2 in the next chapter.

Chapter 3

Previous Results and Related Work

Solving systems of linear equations, being one of the most often performed tasks in scientific computation, enjoys constant devotion of the scientific community. The majority of implementations today use floating point arithmetic on (clusters of) general purpose CPUs, or, more recently, GPUs.

Recent hardware based solvers almost exclusively use floating point arithmetic. Zhang et al. use FPGAs to implement LU decomposition in FP arithmetic without pivoting with the focus on portability among different FPGA platforms [17]. The problem of rounding errors can be addressed by special FP arithmetic and iterative refinement as in [18, 19], or increasign FP precision to Double-Double (128 bits) and Quad-Double (256 bits) [20].

In this chapter we present the summary of previous results on solving systems of linear equations in modular arithmetic, the state-of-the-art of Montgomery multiplication and modular inverse.

3.1 Modular system for solving systems of linear equations

Lórencz and Morháč [1, 2] presented a design of a parallel hardware system for solving dense sets of linear equations precisely. The architecture is presented at fig. 3.1. The modular system uses an array of processing units interconnected together and with a master control unit via buses.

The residual processor uses Gauss-Jordan (GJ) elimination with Rutishauser modification to solve the system of linear congruences. GJ elimination has the advantage of regular memory access pattern and associated regular arithmetic operations. A normal GJ algorithm would produce a reduced row echelon form matrix with a solution of the associated system of linear equations in the last column. We can simplify the GJ algorithm by omitting any row swapping at the expense of having to reorder the solution vector. This GJ algorithm was described in Algorithm 2.1.

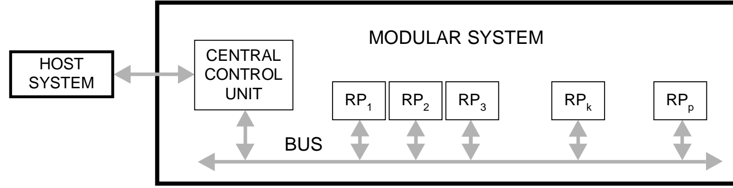


Figure 3.1: Modular system for solving systems of linear equations exactly [1, 2].

Rutishauser modification of Gauss-Jordan elimination discards the unity submatrix produced in the GJ algorithm by shifting the matrix one column to the left as each row is processed. This affects two important operands in Algorithm 3.1, namely the pivot $A_{j,k}$ (step 3) and the element $A_{i,k}$ controling the row reduction in step 5. Both these values will always appear in the first column of the matrix, which simplifies the search for pivot and effectively simplifies the design of the memory interface.

Algorithm 3.1 Rutishauser modification of Gauss-Jordan elimination modulo m.

Input: A is the augmented matrix, n is matrix dimension.
Output: Solution vector $\boldsymbol{x}$ is the first column in A.

1. assuming pivot row j is found during loading of matrix
2. **for** $k = 1..n$ **begin**
3. $\quad A(j, 1..n-k) = |A_{j,1}^{-1}\ A(j, 2..n-k+1)|_m$
4. $\quad$ **for** $i = 1..n$ except j **begin**
5. $\qquad A(i, 1..n-k) = |A(i, 2..n-k+1) - A_{i,1}\ A(j)|_m$
6. $\qquad$ test new pivot, set pivot row j_{new}
7. $\quad$ **end**
8. $\quad j = j_{\text{new}}$
9. **end**
10. **return** $\boldsymbol{x}$ **as** $x_i = A_{i,1}$ **for** $i = 1..n$

3.2 Montgomery multiplication

There are many published papers regarding analysis and implementation of Montgomery multiplication. Most of them concern applications in cryptography, especially RSA. Others deal with Diffie-Hellman key exchange, elliptic curves or focus on the multiplication algorithm itself. We present a summary of some of the published work.

3.2.1 Software approaches

Although we focus on hardware implementation, there are some publications concerning software implementation that are worth mentioning.

Software implementations are designated to run on general-purpose or other processors (vector processors, DSPs). As such, they are specific in that they are limited to fixed word size operations of the particular processor. In order to be able to process long operands, they use multiprecision arithmetic. They often use high-radix algorithms to utilize the word-size granularity of the processors.

A basic adaptation to multiprecision arithmetic is presented in Montgomery's original paper [4]. However, this algorithm is not optimal since it always uses the full length of N' (2.5), which involves more multiplications than necessary. The paper [21] presents a faster way to determine the quotient for modular reduction when we take advantage of the fact that we compute one digit at a time. This is used in Algorithm 2.3, where it suffices to use a single precision arithmetic to compute q using n'_0 (which is also a single digit).

The paper [22] brings an overview of five selected software methods and analyzes their time and memory requirements.

3.2.2 Hardware approaches

There are many hardware implementations of Montgomery multiplication ranging from simple sequential architectures implementing the binary variant of the algorithm to complex unified GF(p) and GF(2^n) pipelined and high-radix architectures.

A basic sequential hardware implementation is sketched in Montgomery's original paper [4] using binary version of the algorithm. Although the paper does not present any hardware architecture, it creates a base on which other authors can build. Several options regarding hardware implementation of modular multiplication are presented in [23].

High-radix implementation on FPGA was presented in [24]. The authors used the Xilinx XC4000 FPGA family. They used a radix of 16 and adopted a technique for simplified quotient determination from [25]. The paper presents also an exponentiation unit and contains a comparison of different configurations used for RSA encryption and decryption. Other high-radix implementations include [26] and [27].

The paper [28] deals with a systolic binary implementation on the Xilinx Virtex V1000-FG680-6 FPGA. The authors present three modifications of the binary Montgomery multiplication algorithm.

Pipelined semi-systolic implementations presented in [29], [30], [31] are targeted at ASIC hardware. The operands are divided into words, resulting in a modified algorithm (Algorithm 3.2). The operand Y, the modulus M and the result S are composed of e words of size w. We add one more word that will be always zero in order to simplify the right-shift operation, since the least significant bit is shifted towards the lower word.

We need the trailing zero word in order to get the last valid word $S^{(e-1)}$ at the end of the inner loop of Algorithm 3.1. Thus $Y = (0, Y^{(e-1)}, ..., Y^{(0)})$, $M = (0, M^{(e-1)}, ..., M^{(0)})$, $S = (0, S^{(e-1)}, ..., S^{(0)})$ and $X = (x_{n-1}, ..., x_0)$, where the words are marked with superscripts and the bits are marked with subscripts.

The architectures presented are unified, i.e. able to compute in both GF(p) and GF(2^n). The architectures are also scalable, providing a means of working with arbitrary-length keys and are configurable in word size and pipeline depth. These architectures were the base

Algorithm 3.2 Multi-word Montgomery multiplication

Inputs: X, Y: operands, M: modulus, n: bit width, w: word size, e: number of words
Output: S: MM(X, Y)
1. $S := 0$
2. **for** $i = 0$ **to** $n - 1$
3. **for** $j = 0$ **to** e
4. $(c_0, S^{(j)}) := S(j) + x_i Y^{(j)} + c_0$
5. **if** $j = 0$ **then** $q := S_0$
6. $(c_1, S^{(j)}) := S^{(j)} + qM^{(j)} + c_1$
7. **if** $j > 0$ **then** $S^{(j-1)}$:= $(S_0^{(j)}, S_{w-1..1}^{(j-1)})$
8. **end for**
9. **end for**
10. **return** S

of one of our experiments. Figure 3.2 shows the pipelined architecture and Figure 3.3 depicts the processing element. The architectures make use of carry-save encoding of the partial product S in order to achieve shorter critical path. The result must be converted to non-redundant binary representation before it can be reused in subsequent multiplications (not shown in the picture).

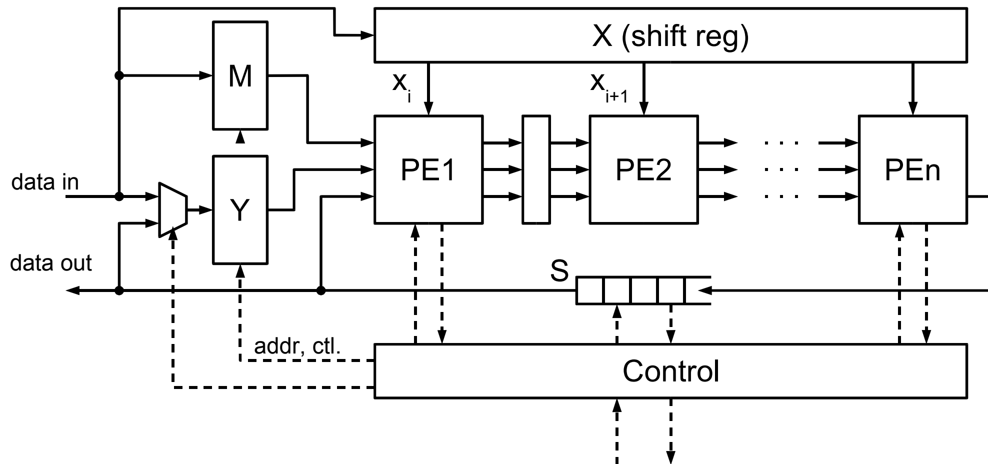


Figure 3.2: Scalable pipelined implementation of the Montgomery multiplier.

Another approach is presented in [32], where both operands and the product are in carry-save form. The architecture is sequential, utilizing full-length adder trees, i.e. the adders are as long as the operands. Two different architectures are presented. They differ in the number of operands in the adder tree. The former one, five-to-two Carry Save Adder (CSA), uses carry-save adder tree to sum all the components. The latter one, four-to-two

Figure 3.3: Pipeline processing element.

CSA, involves smaller adder tree and uses multiplexers and precomputed constants. The variables D and CD are computed at the start of the multiplication and stored in registers.

Algorithm 3.3 Binary Montgomery multiplication with carry-save encoding

---

Inputs: A, CA, B, CB, N, k
Output: $(CS, S) = \mathrm{MM}(CA, A, CB, B)$
1. $(CD, D) := CB + B + N$
2. $(CS, S) := 0$
3. **for** $i = 0$ **to** $k - 1$
4. $\quad q := |CS + S + a_i(CB + B)|_2$
5. $\quad$ **if** $ai = 0$ **and** $q = 0$ **then** $(CX, X) = (0, 0)$
6. $\quad$ **elsif** $a_i = 0$ **and** $q = 1$ **then** $(CX, X) = (0, N)$
7. $\quad$ **elsif** $a_i = 1$ **and** $q = 0$ **then** $(CX, X) = (CB, B)$
8. $\quad$ **else** $(CX, X) = (CD, D)$
9. $\quad$ **end if**
10. $\quad (CS, S) := (CS + S + CX + X)/2$
11. **end for**
12. **return** (CS, S)

3.3 Modular Inverse

There are several papers that describe a hardware implementation of modular multiplicative inverse. The papers [33, 34] present a scalable architecture that deals with inversion of very long numbers that are frequently encountered in cryptography. These operands may be thousands of bits long and are typically stored and processed in a multi-word format. The architecture [34] is designed as unified, i.e. it can perform inversion in GF(p) as well as GF(2^n). A different approach is taken in [35] which uses carry-free arithmetic using a redundant binary encoding using signed digit number representation.

Our efforts focus on shorter numbers (tens to hundreds of bits), which are suitable for error-free computation (and may also be used in elliptic curves cryptography), and therefore we focus mainly on small and fast data paths and controllers. The paper [36] presents a hardware implementation of an ordinary modular inverse using AMI (Almost montgomery inverse, Phase I of MMI) with a modified second phase to obtain MI(a).

3.3.1 Subtraction-Free Montgomery Inverse

The Montgomery modular inverse can be further improved when we focus on the comparison and subtraction operations. Lórencz and Hlaváč in 2005 [37] shown that a negative result of subtraction that would normally have to be discarded can be used without further correction when we exploit the two's complement encoding of the result. This is taken to the extreme so that the u register is always negative and the v always positive, thus the effect of subtraction in the previous inversion algorithms is attained usign addition.

The algorithm thus does not involve any subtractions at all, thus its name — subtraction-free Montgomery modular inverse. The paper [37] focuses only on the first phase of MMI – the Almost Montgomery inverse, see Algorithm 3.4.

3.3.2 Left-Shift Inverse

Modular multiplicative inversion algorithms in GF(p) are often based on the binary extended euclidean algorithm with the operands being compared, subtracted and shifted to the right. However, the modular inversion can also be computed by shifting the operands left, thus aligning them to the most significant bit. The left shifting inversion algorithm was first published in [16]. Recently, this algorithm was used in [38] to design an arithmetic unit for computations in GF(p) with implementation in FPGA.

The algorithm can be split to three parts – the initialization, the main loop which takes a variable number of cycles to complete, and the postprocessing which takes a fixed number of steps.

The algorithm holds five operands in its registers u, v, r, s, p. The u and v registers are used to compute the greatest common divisor of the argument a and the modulus p. Upon initialization u is set to p and v is set to a. The GCD is expected to be 1 for the inverse to exist, but as the values are shifted to the left, the resulting GCD is also shifted and therefore the computed value becomes a power of two. The number of shifts of u is stored

Algorithm 3.4 Subtraction-Free AMI

Input: $a \in [1, p-1]$ and $p > 2$ is prime
Output: $o \in [1, p-1]$ and k, where
$\quad o = a^{-1}2^k \bmod p$ and $n-1 \leq k < 2n$

1. $u \leftarrow (-p), v \leftarrow a, r \leftarrow 0, s \leftarrow 1, k \leftarrow 0$
2. **do**
3.   **if** $(u_{\text{LSB}} == 0)$ **then**
4. $u \leftarrow u/2, s \leftarrow 2s$
5.   **else if** $(v_{\text{LSB}} == 0)$
6. $v \leftarrow v/2,\ r \leftarrow 2r$
7.   **else**
8. $x = u + v,\ y = r + s$
9.   **if** $(CARRY(x) == 0)$ **then**
10. $u \leftarrow x/2,\ r \leftarrow y,\ s \leftarrow 2s$
11.   **else**
12. $v \leftarrow x/2,\ s \leftarrow y,\ r \leftarrow 2r$
13. $k \leftarrow k + 1$

14 **while** $(x \neq 0)$

15. **return** $o \leftarrow s$ and k

in the counter c_u, the number of shifts of v is stored in c_v. Moreover both positive and negative subresults are allowed, so the algorithm will stop as soon as $u = \pm 2^{c_u}$ or $v = \pm 2^{c_v}$. The operations performed on u and v are selected according to the the most significant bits – generally they are shifted to the left whenever possible, otherwise their sign is compared and they are subtracted when their sign is equal or added when their sign is different.

The r and s registers are used to compute the multiplicative inverse. Initialized to $r := 0$ and $s := 1$, the operations with r and s are governed by the values of u and v.

The arithmetic operations involved in the algorithm are $u+v, u-v, v-u, r+s, r-s, s-r$ in table 3.1.

Destination	Initial	Main cycle				Postprocessing			
u	p	$2u$		$u+v$	$u-v$				
v	a	$2v$		$u+v$	$v-u$				
r	0	$2r$	$r/2$	$r+s$	$r-s$	s	$-r$	$p-r$	$r+p$
s	1	$2r$	$s/2$	$r+s$	$s-r$				

Table 3.1: Left-Shift Inverse operation summary

The main loop of the algorithm can be split to the "master" part – computation with u, v, c_u, c_v, and the "slave" part computing with r, s. The "master" part completely

Algorithm 3.5 Left-Shift Inverse algorithm

Input: $a \in [1, p-1]$ and p
Output: $r \in [1, p-1]$, where $r = a^{-1} \bmod p$, 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** $(u_n, u_{n-1} = 1$ & $\mathrm{OR}(u_{n-2}, ..., u_0) = 1)$ **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** $(v_n, v_{n-1} = 1$ & $\mathrm{OR}(v_{n-2}, ..., v_0) = 1)$ **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.

determines the computation of the "slave" part.

The main cycle ends when $u = \pm 2^{c_u}$ or $v = \pm 2^{c_v}$. This can happen either at the beginning when $a = 1$ and thus $v = 1$, or during the computation after addition or subtraction.

Chapter **4**

Overview of Our Approach

Optimal implementation is such an implementation that realizes the algorithm and has an optimum metric in time, area, power consumption or some combination of those. Our optimization goals are time and area, we also analyze the time-area product (sometimes called the quality factor).

The optimization process can done at several levels – from the most general system level downto the physical implementation on silicon. We consider a subset of these levels for our optimization namely the algorithmic level, register transfer level, and gate level. We will follow these basic steps:

- Algorithm selection
- Structure design
- Technology selection
- RTL to gate level synthesis

Algorithm selection. At the algorithmic level, we select the algorithms according to their suitability for hardware implementation. This criterion is the reason for selection of binary variants of the algorithms. We further analyze the selected algorithms with regard to the operations involved in them. We examine the operation count and the data flow in the course of computation. We also analyze the conditions used in branches and loops and the corresponding control flow. We describe the algorithm using a programming language and simulate its behavior in order to gather statistical data about the data and control flows. For high level analysis we use the C/C++ programming language with the NTL [39] library to simulate operations with numbers that do not fit into the machine word size. This analysis serves as a foundation for the first synthesis step which transforms the algorithm to the register transfer level. We use Wolfram Mathematica for verification of correct solution of more complex tasks like solving systems of linear congruences.

Structure design. The next step is to design and analyse the structure of the hardware unit on the register transfer level. The hardware unit at this level consists of blocks

that implement registers, multiplexers, adders, subtractors etc. The types of operations and their occurence in the data flow influence the types and counts of the blocks and their interconnection. Planning together with resource alocation result in the structure of data path and the behavior of the controller that will control the computation. Operand encoding must be considered when implementing arithmetic operations. If specific encoding follows directly from the implementation of an operation, such as the two's complement code after subtraction with negative result, this knowledge can be applied to the original algorithm.

We can describe the hardware unit using a variety of formal methods, including schematics, graphs, tables and hardware description languages (HDLs). We chose a HDL to describe the hardware unit because of the flexibility it can offer when describing parametrized hardware. We have chosen VHDL — other HDLs, such as Verilog, would be equally useful. We use VHDL generics to specify operand length, word length, number of words and other parameters. We describe the hardware units manually at the register transfer level in order to have full control over the architecture while still utilizing the low level optimization and technology mapping capabilities of synthesis tools.

At the RTL level we also perform simulation in order to verify correct function of the described architecture. We write a behavioral VHDL testbench that reads a set of stimuli from a file and exercises the VHDL model of the unit while comparing the outputs with the correct values generated from our C++ and Mathematica programs. We use Mentor Graphics' ModelSim simulator to simulate the model.

Technology selection and RTL to gate level synthesis. The RTL specification is then synthesized to the gate level of the particular hardware platform (FPGA or ASIC). The synthesis tools used for this task are configured using technology libraries.

Each implementation platform has its specific properties that influence the implementation. FPGAs have a more coarse granularity and offer dedicated structures mainly for arithmetic operations such as adders and subtractors, multipliers etc. These dedicated structures can hide some complexity because the cost (area, delay) is lower than when the same function is implemented using general logic. ASICs feature a finer granularity depending on the level of customization. We use a standard cell technology library that provides the functional and quantitative description of elementary logic cells. The synthesis of complex cells such as adders is then controlled by the synthesis library which provides one or more ways to compose a complex cell from elementary cells, e.g. ripple-carry adders or carry look-adead adders.

Another important feature of ASIC synthesis is gate sizing. This is enabled by the technology library containing several cells with same function differing in the size of their internal transistors, thus with different delay and area values. The synthesis tool can then trade delay for area by selecting the appropriate cell variation.

Memory arrays require special attention because ASIC synthesis tools cannot infer memory elements other than registers form the standard cell library. In order to use on-chip memories, we need technology libraries that come with memory compilers. We use the memory compiler to create memory blocks of appropriate size and then we can use (instantiate) them in our VHDL design.

The synthesis process results in a gate level netlist of the designed circuit. This model is more detailed than the source RTL VHDL model because it is described using technology cells from the library. Design area and delays are estimated from the netlist and can be annotated to the individual gates and signals in the circuit. Because the operations are performed on individual bits of the operands, each bit can have a different delay. This information can be utilized on the RTL or even algorithmic level to further modify and optimize the algorithm and its architecture.

We synthesize the hardware unit to two different technologies. First target technology is the Xilinx FPGA, for example Virtex2 (xc2v4000) or Virtex6 (xc6vsx475t). We use Xilinx ISE toolchain with XST for synthesis. We synthesize the unit and perform placement and routing. The time and area data is extracted from the placed and routed result. Second target technology is 0.13μ ASIC (TSMC or GlobalFoundries) with Synopsys Design Compiler 2005.09 SP2 (RTL synthesis). The time and area data is a post-synthesis estimate extracted from the synthesized netlist using time and area reporting in Design Compiler.

We perform synthesis of several variants of the architecture parametrized to several different operand lengths and word lengths, where applicable. For a selected subset of models we also perform post-place and route and post-synthesis simulation in order to verify the correct function of the resulting detailed model. For the FPGA technology we perform post-place and route simulation using Simprim simulation libraries and ModelSim. For the ASIC technology we perform post-synthesis simulation using a simulation library converted form the synthesis library using Synopsys Library Compiler and ModelSim.

We present two modifications of existing architectures for mongomery multiplication, both implemented in FPGA. The former one is a pipelined multiplier, where we study the effect of the underlying FPGA architecture on variable word size and pipeline depth. The latter one is based on a sequential carry-save architecture where both the operands and the partial product are encoded in carry-save form. We modify the carry-save encoding in that we reduce the redundancy and take advantage of the fast dedicated carry chains in the FPGA.

Among various algorithms for computing the multiplicative modular inverse we have selected two, montgomery modular inverse and left-shift inverse. We focus on optimizing these two algorithms and their variants and comparing their relative implementation metrics in FPGA and ASIC. The algorithms were selected because of their suitability for use in GF(p) and our previous experience. Montgomery inverse is frequently used and its variants and associated architectures are under constant development. The Subtraction-Free algorithm for the Almost Montgomery Inverse [37] was recently published at the time of our research. The Left-Shift Inverse algorithm [16] for the classical modular inverse was promising because of the low number of subtractions needed to compute the inverse. Both variants were originally studied for the purpose of implementing cryptographic algorithms, they are however also suitable for solving systems of linear congruences.

Finally we implement a dedicated hardware processor for solving systems of linear congruences in FPGA and ASIC. We use internal on-chip memory in the form of BLOCKRAM in FPGA and synchronous static memory block compiled using special memory compiler in ASIC.

4.1 FPGA implementation of a pipelined Montgomery multiplier

The multi-word Montgomery multiplication algorithm can be implemented on various architectures in software and in hardware. The architecture consists of a computing pipeline of p processing elements. Each processing element (PE) consists of 2 partial product generators, 2 adders and a shift and alignment layer, which includes registers. The latency of a single PE is 1 clock cycle. The PEs are separated by registers, therefore the latency of one pipeline stage is 2 clock cycles.

The Y operand and the modulus M are stored in memory. The X operand is stored in a shift register, since after x_i was used in the computation, it is no longer needed and can be discarded. The partial results coming out of the pipeline are stored in a queue until they can be used again in the computation.

Our architecture [A.13] differs from [29] in that we do not use redundant carry-save form of the intermediate result, because the underlying FPGA architecture includes dedicated carry logic and interconnect. Therefore a binary adder that uses FPGA-specific carry chain is faster than a carry-save adder, as long as the carry chain is not split into pieces, which would involve additional delay since the general-purpose routing is much slower than the dedicated carry interconnection.

4.2 Montgomery multiplier with modified Carry-Save encoding

We consider implementation on a Field Programmable Gate Array (FPGA) reconfigurable hardware.

Our improvement of the architecture [32] is based on the fact, that in an FPGA, there is hardware dedicated for implementing fast ripple-carry adders. We can make use of it when considering the possible encoding of the operands.

Instead of using normal carry save encoding, we modify the encoding such that a carry bit is "saved" only for each w-th bit. We break the operands into e words of length w, $e = k/w$. If $w = 1$, we get the conventional carry-save encoding. If $w > 1$, we get a modified (relaxed) carry-save encoding (Figure 4.1). Our approach was published in [A.1].

We can use both the conventional and the modified carry-save representation to construct adder trees. Examples of four-to-two carry-save adder trees are presented in Figure 4.2.

When using conventional carry-save encoding, we get $(CS, S) = \mathrm{CSA}(A_1, A_2, A_3, A_4)$, where S is the sum, CS is the carry and operands A_i as well as S and CS have the same width.

Using modified carry-save encoding we have $(CS, S) = \mathrm{CSA}'(A_1, CA_1, A_2, CA_2)$, where the sum parts A_1, A_2 and S are k-bit numbers, whereas the carry parts CA_1, CA_2 and CS are e bits long.

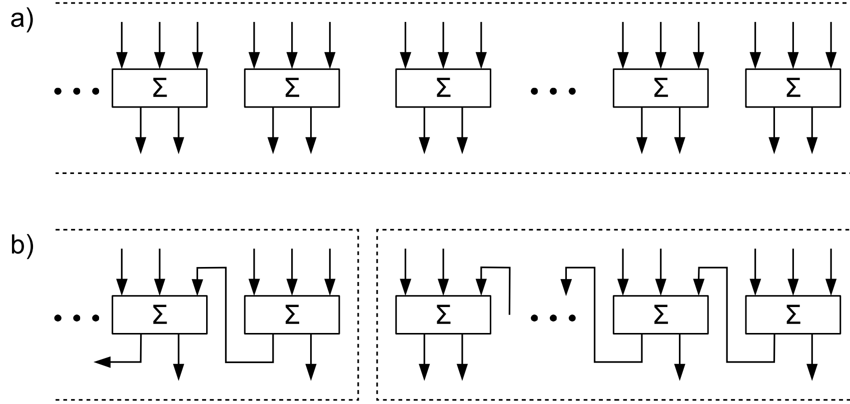


Figure 4.1: Internal structure of adders: a) conventional carry-save adder, b) modified carry-save adder.

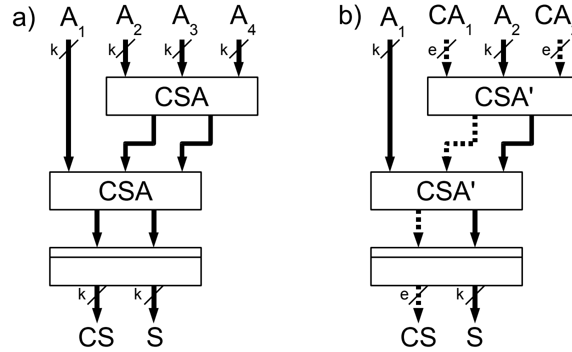


Figure 4.2: Adder tree using a) carry-save encoding, b) modified carry-save encoding.

The architecture is derived from [32], we consider four-to-two CSA. It consists of two layers of carry-save adders with k-bit operands. We substitute w-bit sections of the carry-save adder with sections of ripple-carry adders. Consequently, in each word, $w-1$ bits of the carry part (CS) are zeros, thus do not need to be stored in registers (Figure 4.3).

For $w > 1$, we extend the carry chain fragments, thus extending also the critical path. However, due to the fast dedicated adders in FPGA, this needs not necessarily to introduce significant delay to the critical path.

In the implementation phase the automatic synthesis, placement and routing tools make use of the dedicated adder hardware present in FPGA and construct several carry chains of length $2w$. The carry chains consist of one adder from the first layer and one adder from the second layer (Figure 4.3).

When increasing the word size w, we expect the occupied area to decrease, because the number of carry bits decreases and we save more registers and logic cells. We expect that the additional delay introduced by the ripple-carry adder segments is compensated by the savings in routing resources associated with the carry bits.

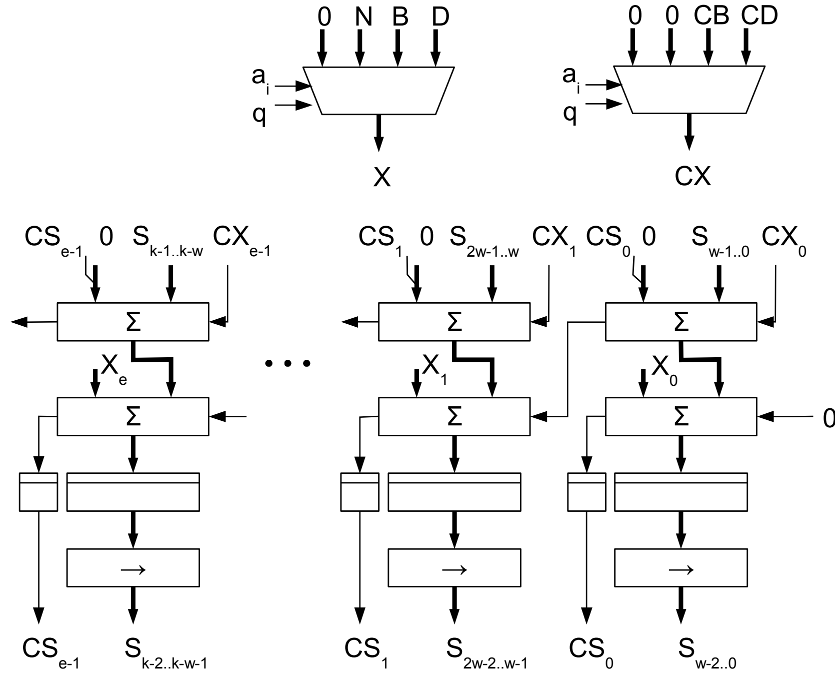


Figure 4.3: Multiplier architecture with modified CSA adder tree.

4.3 HW optimization of Left-Shift Inverse

The left-shift inversion algorithm (Algorithm 3.5) can be modified to better utilize hardware components, as is shown in Algorithm 4.1. The main area of changes is in the way the test conditions are computed. For example, step 3 of Algorithm 3.5, $u \neq \pm 2^{c_u}$ & $v \neq \pm 2^{c_v}$ can be implemented by masking using an auxiliary register m.

Consider an example using 5 bits, where $c_u = 2$, and we are testing whether $u = \pm 2^{c_u}$. The binary image of $2^{c_u} = 2^2$ is 00100, whereas for -2^2 it is 11100. Two least significant bits must be zero. In this case only one mask value is sufficient to test both $\pm 2^2$: Let $m = 11100$. Then $u = -2^2$ can be tested with a simple equality comparator. The other case of $u = 2^2$ can be tested by masking with shifted value of m. If u & $(m \ll 1) = 000000$ then $u = 2^2$. This follows from the fact that u was shifted 2 bits left and so the 2 least significant bits must be zero.

Similarly, the counter comparison tests in steps 10 and 19 of Algorithm 3.5 can be optimized using flags d and $u/\bar{v}$. This way we can replace the expensive "$\leq$" and "$\geq$" test with a simpler equality test and sequential tracking of counter values. This approach is reflected in the HW optimized Algorithm 4.1.

Algorithm 4.1 Left-shifting inversion algorithm, HW optimized variant

```
Input: a ∈ [1, p − 1] and p
Output: r ∈ [1, p − 1], where r = a^−1 mod p, c_u, c_v
        and 0 < c_v + c_u ≤ 2n
1.  u := p, v := a, r := 0, s := 1, c_u = 0, c_v = 0, u/v̄ := 1, m := 1, t_pos := 1, t_neg := 1
2.  while (t_neg = 1 & t_pos = 1)
3.      if (u_n, u_{n−1} = 0) or (u_n, u_{n−1} = 1 & OR(u_{n−2}, ..., u_0) = 1) then
4.          if (d = 0 or u/v̄ = 0) then
5.              u := 2u, r := 2r, c_u := c_u + 1, u/v̄ := 0
6.          else
7.              u := 2u, s := s/2, c_u := c_u + 1
8.      else if (v_n, v_{n−1} = 0) or (v_n, v_{n−1} = 1 & OR(v_{n−2}, ..., v_0) = 1) then
9.          if (d = 0 or u/v̄ = 1) then
10.             v := 2v, s := 2s, c_v := c_v + 1, u/v̄ := 1
11.         else
12.             v := 2v, r := r/2, c_v = c_v + 1
13.     else
14.         if (v_n = u_n) then
15.             if (d = 0) then
16.                 x := u − v, u := 2x, r := 2(r − s), c_u := c_u + 1, u/v̄ := 0, wu := 1
17.             else if (u/v̄ = 1) then
18.                 x := u − v, u := 2x, r := r − s, s = s/2, c_u := c_u + 1, m := 2m, wu := 1
19.             else
20.                 x := v − u, v := 2x, s := s − r, r = r/2, c_v := c_v + 1, m := 2m, wu := 0
21.         else
22.             if (d = 0) then
23.                 x := u + v, u := 2x, r := 2(r + s), c_u := c_u + 1, u/v̄ := 0, wu := 1
24.             else if (u/v̄ = 1) then
25.                 x := u + v, u := 2x, r := r + s, s = s/2, c_u := c_u + 1, m := 2m, wu := 1
26.             else
27.                 x := v + u, v := 2x, s := s + r, r = r/2, c_v := c_v + 1, m := 2m, wu := 0
28.         t_pos := OR(x & 2m), t_neg := OR(x ⊕ m)
29.     d := OR(c_u ⊕ c_v)
30. if (wu = 1) then
31.     if (u/v̄ = 1) then
32.         s := r
33.     else
34.         s := r/2
35. if (t_neg = 0 & s_m = 1 or t_pos = 0 & s_m = 0) then
36.         r := 0
37.     else
38.         r := p
39. if (t_neg = 1) then
40.     r := r − s
41. else
42.     r := r + s
43. return r
```

The original table of operations (Table 3.1) is thus modified resulting in Table 4.1.

Destination	Initial	Main cycle						Postprocessing			
u	p	$2u$		$2(u+v)$	$2(u-v)$						
v	a	$2v$		$2(u+v)$	$2(v-u)$						
r	0	$2r$	$r/2$	$2(r+s)$	$2(r-s)$	$r+s$	$r-s$	0	p	$r-s$	$r+s$
s	1	$2r$	$s/2$	$2(r+s)$	$2(s-r)$	$r+s$	$s-r$	r	$r/2$		

Table 4.1: Left-Shift Inverse HW optimized operation summary

The computation unit consists of three major parts – datapath, controller and I/O block. The datapath contains data memory elements (registers) to store the operands and intermediate values (u, v, r, s, p, m), two adders/subtracters to compute the arithmetic operations, and multiplexers that switch the cata flow according to the particular operation to be performed. The controller contains a finite state machine, two counters and some logic, whose role is to control the progress of computation using control and status signals to the datapath. The I/O block connect the internal input and output buses to the external bidirectional bus and contains associated control and status signals of the whole unit.

4.3.1 Datapath

The datapath is depicted in Figure 4.4. It contains two adders/subtractors. ADD1 is a reversible adder/subtracter, i.e. it performs $u+v$, $u-v$ or $v-u$. Subtraction is performed using bitwise negation of one of the operands and presenting a "hot one" carry-in. Each of the inputs of the adder can be negated separately, thus determining order of subtraction $(u-v$ or $v-u)$.

4.3.2 Controller

The controller controls the behavior of the inversion unit. The central unit is the finite state machine (FSM) that receives status signals and produces control signals that are connected to the datapath and I/O block. Also contained in the controller are two counters that hold the values of c_u and c_v and an inequality comparator that compares their values. Furthermore, the controller contains flag registers that hold values of test logic functions from the data path.

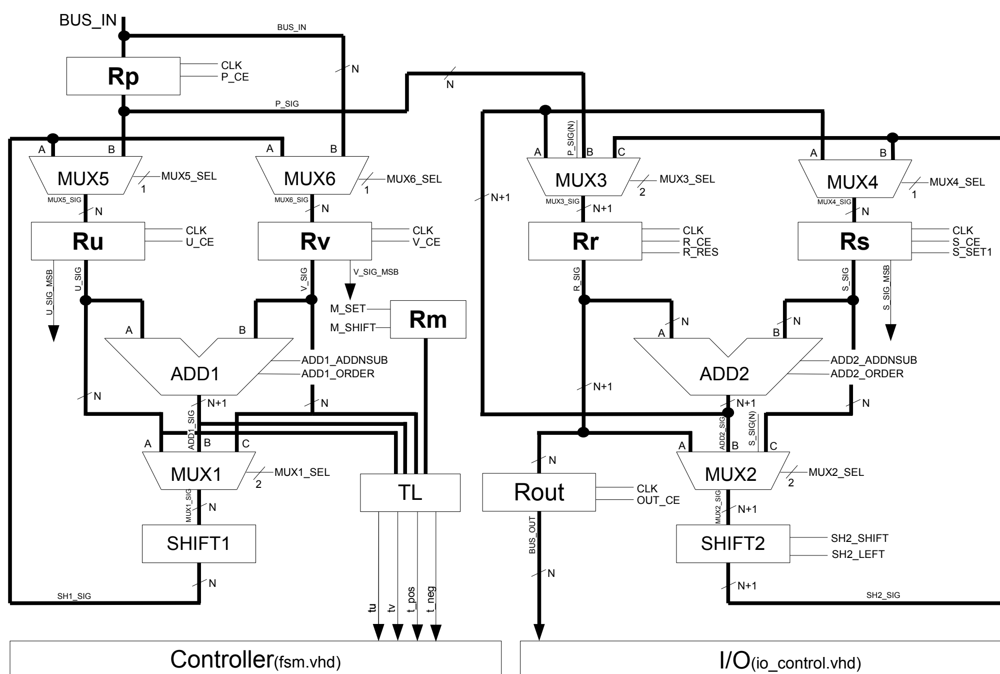


Figure 4.4: LSI datapath schematic

4.4 Comparing Montgomery inverse architectures

We implement three main variants of hardware architectures for Montgomery modular inverse.

When implementing MMI in hardware, we can divide the data path into the master part that computes $\gcd(u, v)$, and the slave part that computes the inverse (uses r, s). We have studied three different hardware units for computing MMI, which differ in the implementation of the master part of the data path as depicted in fig. 4.5. We have also implemented the corresponding controllers.

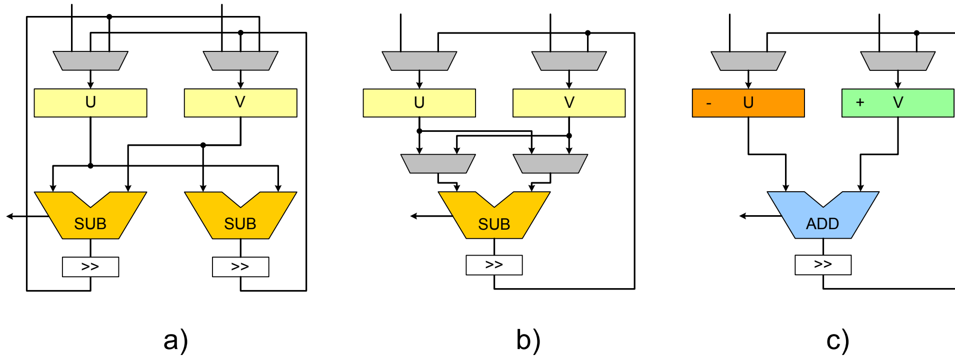


Figure 4.5: Master part of AMI: a) Two subtractors, b) one subtractor with swappable inputs, c) subtraction-free.

The simplified schematic in fig. 4.5 only shows the master part that are needed for the first phase of MMI, i.e. the Almost Montgomery inverse. The slave part is not shown for simplicity. The comparison of these architectures was published in [A.3].

These include AMI with one subtractor and AMI with two subtractors, which implement the first phase of Algorithm 2.7, and Subtract-free AMI, which implements Algorithm 3.4. The slave part of the data path is always the same, it contains one adder for computing $(r + s)$.

Variants of implementation of Algorithm 2.7 involve computations of both $(u-v)$ (step 5) and $(v-u)$ (step 6). There are two possible approaches to this requirement – extra time or extra space (chip area).

The first approach leads to AMI with one subtractor, which uses an extra clock cycle to compute the opposite difference on the same subtractor as the first difference. The master part contains one subtractor to compute $x = (u-v)$. The output x of the subtractor is then used for the test $(u > v)$, and if the condition is true, x is used in subsequent computation. If the test condition is false (x is negative), the same subtractor is used again in the next clock cycle to compute the opposite difference $x = (v - u)$.

The second approach, AMI with two subtractors, uses the two subtractors to compute both differences concurrently. Therefore both differences, $(u - v)$ and $(v - u)$, are always available. The area is larger due to the additional subtractor, but there are no additional

clock cycles, therefore this implementation is faster than AMI with one subtractor. The number of clock cycles needed to compute the inverse corresponds to the number of iterations of the algorithm main loop.

The subtraction-free AMI implementation contains one adder in the master part, and one adder in the slave part. The result of addition $x = (u + v)$ is always divided by 2 (shifted right), and written into either u or v; the destination register determined by the sign of the result – this information is contained in carry.

The resulting architecture computes AMI in the same number of cycles as AMI architecture with two subtractors, but it is considerably smaller (contains 1 less adder). It is even possible to be slightly faster due to more simple logic (simpler multiplexers, no need to switch between $(u - v)$ and $(v - u)$).

4.5 Solver of systems of linear equations

From the upper bound for M, which follows from the Hadamard's inequality (2.3), a simplified expression can be derived as $M > 2n^{\frac{n}{2}}B^n$, where $B = \max(a_{ij})$. We can estimate the the number of bits of M by taking a binary logarithm, therefore $\log_2 M > \log_2(2n^{\frac{n}{2}}B^n) = \mathrm{O}(n \log B + n \log n)$. The number of bits of M together with the length of individual moduli (the machine word length) determines the number of moduli needed for the RNS to represent the solution of the SLE (2.1).

The machine word size e will be selected to accommodate all required individual moduli m_i. Because of the potentially large number of moduli and the requirement for them to be mutually coprime, they should be prime numbers. This has further advantage that any nonzero number is invertible, which simplifies the search for a pivot during Gauss-Jordan elimination.

Papers [1] and [2] design a hardware RNS linear equation solver — Modular System (MS), depicted in Figure 3.1. However, an implementation of this system was very difficult at that time. With current technologies, it is possible to implement the system, and especially FPGA technologies offer a straightforward implementation with reconfiguration possibilities based on the cardinality of the problem and optimize for time and area.

The error-free solution of an SLE with operations performed in residue arithmetic is implemented in this special MS. The MS typically has a parallel SIMD architecture, and consists of a control unit and several identical processing units – (RP)s denoted as RP_1, RP_2, ..., RP_p interconnected with a BUS.

Individual residual processors compute the solution of a linear system in the modulus m_k assigned to RP_k independently of the others. Due to the fundamental properties of RNS, the addition is carry-free, subtraction borrow-free across the individual moduli, and therefore the computation can occur safely in parallel. Once the computation is done, the result is transformed back into the rational number set either with the Chinese Remainder Theorem or the Mixed Radix Conversion.

The architecture of the residual processor RP_k is depicted in Fig. 4.6 consisting of Memory, Arithmetic Units AU_2, AU_3, ..., AU_{n+2} and the Control unit.

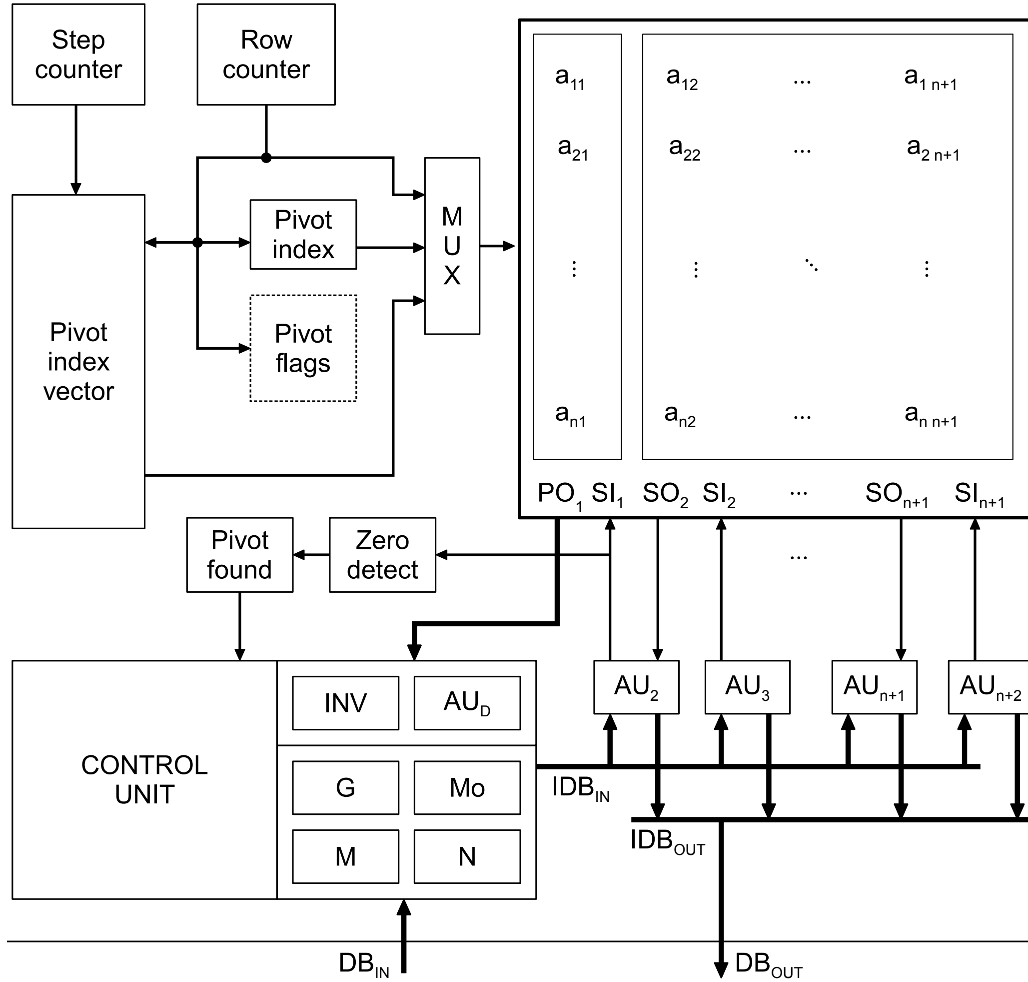


Figure 4.6: Arhitecture of the residual processor

In [A.4], we present a FPGA implementation of the residual processor. For an efficient FPGA implementation, several parts of the system must be redesigned to use resources found in modern FPGAs. This is true especially for the memory architecture, which in [2] used asynchronous logic and custom memory elements. We present a new memory architecture using standard RAM blocks found in most recent FPGAs. We also redesigned the addressing and pivoting logic to support efficient implementation of the elimination algorithm used.

In [A.5], we extend the implementation to ASIC technology in 130 nm standard cells, and compare area and time performance of individual components as well as the complete residual processor. For internal memory, we use compiled synchronous static memory blocks available with ASIC standard cell library.

In [A.6], we compare the ASIC implementation in three different standard cells libraries,

namely Synopsys/GlobalFoundries 130 nm and Faraday/UMC 110 nm and 55 nm. The 130 nm and 110 nm libraries are high performance libraries, while the 55 nm library is a low power library. For the internal memory, we use the memory compiler avaliable in each library. The results show that considering a suitable die size around 1 cm^2 , the maximum matrix dimension is 1000 for the 130 and 110 nm technologies, and 2000 for the 55 nm low power technology.

In [A.7], we integrate the Residual Processor as part of a System on Chip architecture on an FPGA with an integrated CPU and peripheral interconnect.

In [A.8], we bring together the previous results and create the modular system that performs all the tasks necessary to solve the systems of linear congruences and prepare the solutions in individual moduli of the RNS. Together with the host system, the final solution of the system of linear equations is computed (as a rational number or to any desired precision).

Chapter 5

Author's Relevant Papers

This chapter presents the main results of our research. In the following six sections, a collection of our papers is presented as they were published in the coresponding conference proceedings or journal. Each paper is preceded by a short introduction in order to set it in context.

The following papers are included:

- **RP1** [A.1] *Montgomery Multiplication on FPGA with Modified Carry-Save Encoding.* This paper examines a modification of the carry-save encoding scheme for the purpose of optimizing a FPGA implementation of a Montgomery modular multiplier.
- **RP2** [A.3] *Comparing Subtraction-Free and Traditional AMI.* This paper compares three architectures for Almost Montgomery Inverse modular inverse differing in the type and number of arithmetic operations used. Implementation results in FPGA are compared.
- **RP3** [A.4] *Dedicated Hardware Implementation of a Linear Congruence Solver in FPGA.* This paper details an FPGA implementation of a solver of systems of linear congruences.
- **RP4** [A.5] *Comparison of FPGA and ASIC Implementation of a Linear Congruence Solver.* Evolving from **RP3**, the SLC solver is adapted to an ASIC technology with memory compiler and the results are compared.
- **RP5** [A.6] *An ASIC Linear Congruence Solver Synthesized with Three Cell Libraries.* The SLC solver from **RP3** and **RP4** is implemented in three different standard cell libraries comparing the results in order to estimate attainable size and performance.
- **RP6** [A.8] *Design of a Residue Number System Based Linear System Solver in Hardware.* This paper presents a system for solving Linear system solver in hardware using the SLC solvers presented in **RP3-5**.

The following figure 5.1 illustrates the relationships among individual papers presented

in this chapter.

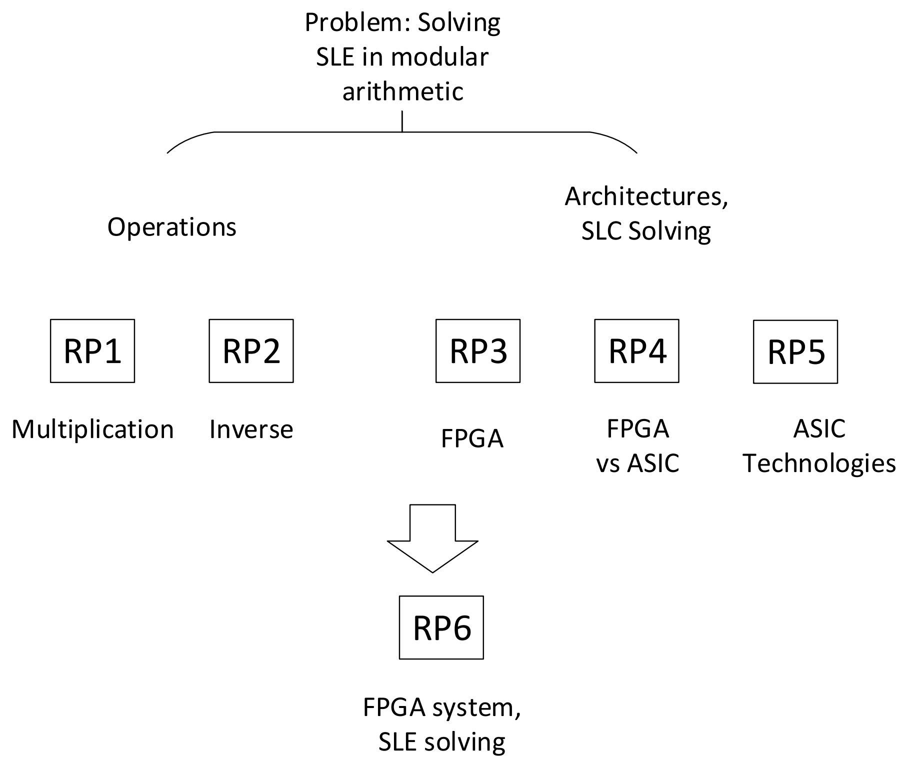


Figure 5.1: Relationships among author's relevant papers.

5.1 RP1 – Montgomery Multiplication on FPGA with Modified Carry-Save Encoding

Modular multiplication is one of the main operations in Gauss-Jordan elimination for solving systems of linear congruences. It is also used heavily in cryptography, for example in RSA or elliptic curve cryptography. Although **RP1** was originally written mainly with cryptography in mind, we feel it is still relevant for the purpose of solving linear systems and it is important to examine possible optimization of this operation on FPGA.

The paper was published in *International Conference on Signals and Electronic Systems, ICSES'04, 2004* [A.1]. This paper deals with optimization of hardware Montgomery multiplication units implemented in FPGA. FPGAs typically contain dedicated structures for implementation of arithmetic operations. This work focuses on efficient usage of dediceted carry chain logic for implementation of carry-save encoded Montgomery multipliers.

Traditionally, carry-save encoding aims to circumvert the problem of long carry chains increasing the critical path delay of artithmetic units by saving the carry bits for later clock cycles. In FPGAs, the same problem is solved by introducing dedicated carry chain logic structures that are much faster than the generic logic and interconnection fabric. Our approach is to combine the two aspects by modifying the carry save encoding to save less bits, effectively trading the dedicated carry chain length against the amount of carry-save bit logic. Our architecture is based on [32], and we analyze the impact of saving less carry bits on the time and area of the multiplication unit.

Montgomery Multiplication on FPGA with Modified Carry-Save Encoding

J. Buček, R. Lórencz

Department of Computer Science and Engineering, Faculty of Electrical Engineering,
Czech Technical University, Karlovo náměstí 13, 12135 Praha 2, Czech Republic
e-mail: bucekj@fel.cvut.cz, lorencz@fel.cvut.cz

Abstract **– The modular multiplication is the core operation and also the most time consuming operation in many cryptographic applications such as RSA and elliptic curve cryptography. We present a modification of an architecture for the Montgomery algorithm for modular multiplication in GF(p) on a Field Programmable Gate Array (FPGA). Montgomery multiplication is widely used because it can be implemented in faster hardware than the conventional modular multiplication.**

Keywords: Montgomery multiplication, GF(p), FPGA

I. Introduction

As the demand for secure computing and secure communication grows, cryptographic applications play a more important role than ever. The cryptographic algorithms used to implement the security properties are often based on modular arithmetic operations, particularly modular exponentiation which is achieved by repeated modular multiplication.

The Montgomery multiplication algorithm is often used to implement these operations. As opposed to the conventional modular multiplication, the Montgomery's algorithm [1] does not use division by the modulus N, rather it uses division by 2^i, which is faster, since it is done merely by shifting (i is an integer).

Moreover, the reduction step is performed depending on the least significant bit (LSB) of the intermediate result, rather than on the MSB, as in the case of a conventional multiply. This causes another speedup, since the critical path does not necessarily go through the full carry chain of the adder.

Montgomery multiplication operates on the set of images (the Montgomery domain). The Montgomery image of the k-bit number a is defined as $\overline{a} = a\,R \bmod N$ where $R = 2^k$, $a < N < R$ and N is an k-bit number relatively prime to R. The Montgomery multiplication computes (1).

$$\overline{c} = \mathrm{MM}(\overline{a}, \overline{b}) = \overline{a}\,\overline{b}\,R^{-1} \bmod N \quad (1)$$

It can be shown that $\overline{c}$ is the image of $c = ab$, since

$$\begin{aligned} \overline{c} &= \overline{a}\,\overline{b}\,R^{-1} \bmod N \\ \overline{c} &= a\,b\,R \bmod N \\ \overline{c} &= c\,R \bmod N \end{aligned} \quad (2)$$

Assuming that $A = \sum_{i=0}^{k-1} a_i 2^i$ and $S = \sum_{i=0}^{k-1} s_i 2^i$, we may describe the basic binary Montgomery multiplication algorithm as presented in Algorithm 1.

Algorithm 1. Binary GF(p) Montgomery multiplication
Inputs: A, B, N, k
Output: S = MM(A, B)

```
S := 0
for i = 0 to k-1
      S := S + a_i B
      S := S + s_0 N
      S := S/2
end for
```

II. Previous Work

There are applications, particularly in cryptography, where long operands are common (for example, more than 512 bits). In order to achieve sufficient speed of the multiplication, one has to find a way around the long carry path implied by the adders. There are several approaches to this problem [2,3,4,5].

One possible method divides the operand into words and treats them separately, splitting the carry chain by registers and thus implementing a pipeline. Another possible method eliminates the carry chain by using redundant Carry Save (CS) encoding of the intermediate result S.

These methods can be combined together [4]. In order to implement more complex computation, such as a modular exponentiation, it is necessary to reuse the result of one multiplication in subsequent multiplications. Hence, it is desirable to avoid unnecessary conversions between redundant and non redundant forms of any of the operands.

The architecture proposed at [6] solves this by leaving both operands A, B as well as the result S in CS form. This approach ensures high throughput of an RSA exponentiation because the intermediate results can be directly used as operands of the next multiplication.

III. Our Approach

We consider implementation on a Field Programmable Gate Array (FPGA) reconfigurable hardware.

Our improvement of the architecture [6] is based on the fact, that in an FPGA, there is hardware dedicated for implementing fast ripple-carry adders. We can make use of it when considering the possible encoding of the operands.

Instead of using normal carry save encoding, we modify the encoding such that a carry bit is "saved" only for each w-th bit. We break the operands into e words of length w, $e = k/w$. If $w = 1$, we get the conventional carry save encoding. If $w > 1$, we get a modified (relaxed) carry save encoding (Fig. 1).

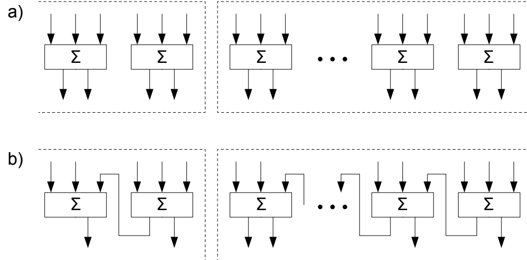


Fig. 1: a) carry save adder, b) modified carry save adder.

We can use both the conventional and the modified carry save representation to construct adder trees. Examples of four-to-two carry save adder trees are presented in Fig. 2.

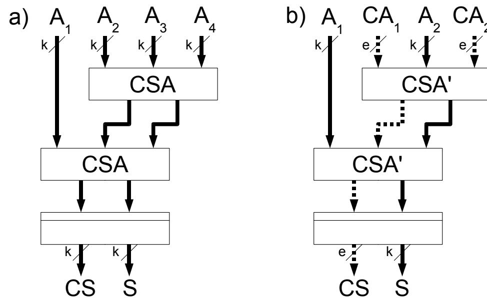


Fig. 2: Adder tree using a) carry save encoding, b) modified carry save encoding.

When using conventional carry save encoding, we get $(CS,S) = \mathrm{CSA}(A_1,A_2,A_3,A_4)$, where S is the sum, CS is the carry and operands A_i as well as S and CS have the same width.

Using modified carry save encoding we have $(CS,S) = \mathrm{CSA}'(A_1,CA_1,A_2,CA_2)$, where the sum parts A_1, A_2 and S are k-bit numbers, whereas the carry parts CA_1, CA_2 and CS are e bits long.

Algorithm 2. Binary GF(p) Montgomery multiplication with carry save encoding
Inputs: A, CA, B, CB, N, k
Output: (CS,S) = MM(CA, A, CB, B)

```
(CD, D) := CB + B + N
(CS, S) := 0
for i = 0 to k-1
    q := CS + S + a_i (CB + B) mod 2
    if a_i = 0 and q = 0 then (CX,X) = (0,0)
    elsif a_i = 0 and q = 1 then (CX,X) = (0,N)
    elsif a_i = 1 and q = 0 then (CX,X) = (CB,B)
    else (CX,X) = (CD,D)
    end if
    (CS, S) := ( CS + S + CX + X ) / 2
end for
```

The architecture is derived from [6], we consider four-to-two CSA. It consists of two layers of CS adders with k-bit operands. We modify the architecture in such a way that we substitute w-bit sections of the carry-save adder with fragments of ripple-carry adders. Consequently, in each word, w-1 bits of the carry part (CS) are zeros, thus do not need to be stored in registers (Fig. 3).

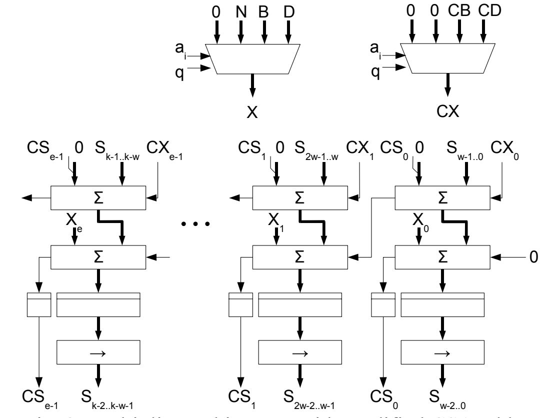


Fig. 3: Multiplier architecture with modified CSA adder tree

For $w > 1$, we extend the carry chain fragments, thus extending also the critical path. However, due to the fast dedicated adders in FPGA, this needs not necessarily to introduce significant delay to the critical path.

In the implementation phase the automatic synthesis, placement and routing tools make use of the dedicated adder hardware present in FPGA and construct several carry chains of length $2w$. The carry chains consist of one

adder from the first layer and one adder from the second layer (Fig. 3).

When increasing the word size *w*, we expect the occupied area to decrease, because the number of carry bits decreases and we save more registers and logic cells.

We expect that the additional delay introduced by the ripple-carry adder segments is compensated by the savings in routing resources associated with the carry bits.

IV. Implementation

The architecture was described in VHDL, simulated and synthesized to an FPGA.

The VHDL description was performed using generic constants for operand size *k* and word size *w*. Resulting code was synthesized using Synplicity Synplify Pro. The synthesized netlist was then placed and routed using Xilinx ISE 5.2i.

The placement and routing was done automatically. The desired clock frequency was set at 80 MHz and the implementation process was run multiple times, adjusting the clock frequency constraint each time in order to obtain the best result possible for the given configuration.

V. Results

We have implemented the multiplier with configurable operand and word size. The implementation results for operand size $k = 512$, 1024 and 2048 are presented below.

Table 1. Implementation results for $k = 512$

Word size (*w*)	Area (slices)	Max Frequency (MHz)	Time x Area
1	4633	147,08	31,50
2	4231	121,30	34,88
4	**3644**	**116,66**	**31,24**
8	3453	99,92	34,56

Table 2. Implementation results for $k = 1024$

Word size (*w*)	Area (slices)	Max Frequency (MHz)	Time x Area
1	9328	129,8	71,86
2	9210	116,5	79,06
4	**7373**	**113,6**	**64,9**
8	7088	92,83	76,35
32	6574	87,82	74,86
64	6487	64,6	100,43
128	6437	48,91	131,62

Table 3. Implementation results for $k = 2048$

Word size (*w*)	Area (slices)	Max Frequency (MHz)	Time x Area
1	18873	114,65	164,61
2	18086	105,24	171,85
4	15384	94,34	163,07
8	**14280**	**89,53**	**159,49**
16	13436	80,19	167,56

The area occupation decreases with increasing word size *w*. The time-area product has its minimum in $w = 4$ for $k = 1024$ and 512, and in $w = 8$ for $k = 2048$. For the minimum time-area product, the area saving more than 20% relatively to the conventional carry-save encoding ($w = 1$). In Table 1 to 3, the minimum time-area product is shown in bold.

Our architecture slows down the clock frequency, as the prolonged carry chains enlarge the critical path delay. The savings in routing resources have lower effect than expected, because in the original CSA design, the routing paths were mostly local.

VI. Conclusions

We have implemented the Montgomery modular multiplication in FPGA using modified carry save encoding. By utilizing the dedicated carry logic in the FPGA, we have been able to save circuit area. The results show a decrease in occupied area for word size $w > 1$. The clock frequency also decreases, however, we can find a minimum time-area product for $w > 1$ while saving more than 20% area.

References

[1] P. L. Montgomery, "Modular Multiplication without Trial Division", *Math. of Computation*, vol. 44, pp. 519-521, Apr. 1985.

[2] Alexandre F. Tenca, Çetin K. Koç, "A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm", *IEEE Transactions on Computers*, Vol. 52, No. 9 (September 2003), pp.1215-1221

[3] Çetin Kaya Koç, Tolga Acar, Burton S. Kaliski, Jr., "Analyzing and Comparing Montgomery Multiplication Algorithms", *IEEE Micro*, Volume 16, Issue 3 (June 1996), pp.26-33

[4] E. Savas, A.F. Tenca, C. K. Koc, "A Scalable and Unified Multiplier Architecture for Finite Fields GF (p) and GF(2m)", *Proc. Second Int'l Workshop Cryptographic Hardware and Embedded Systems—CHES 2000*, C. K. Koc¸ and C. Paar, eds., pp. 277-292, Aug. 2000.

[5] T. Blum, C. Paar, "High-Radix Montgomery Modular Exponentiation on Reconfigurable Hardware", *IEEE Transactions on Computers*, Vol. 50, No. 7 (July 2001), pp. 759-764

[6] C. McIvor, M. McLoone, J. McCanny, A. Daly and W. Marnane, "Fast Montgomery Modular Multiplication and RSA Cryptographic Processor Architectures", *37th Asilomar Conference on Signals, Systems, and Computers,* Nov 2003

5.2 RP2 – Comparing Subtraction-Free and Traditional AMI

Another important operation when solving systems of linear coruences is the modular inverse. It is used in each elimination step to compute the multiplicative inverse of the pivot, a value used subsequently in multiplication to process the remaining elements in the pivot's row.

This paper was published in *Design and Diagnostics of Electronic Circuits and systems, DDECS'06, 2006* [A.3]. The paper deals with FPGA implementations of multiplicative inverse operations in GF(p). More precisely, it compares three variants of the Almost Montgomery Inverse (AMI), an operation that computes $|a^{-1}2^k|_n$, and is the first part of the computation of the Montgomery inverse. The variants studied are the Subtraction-Free AMI, and the classical AMI with subtractions in two variants (one and two subtractors).

The relatively new Subtraction-Free algorithm makes use of the fact that in GCD computation, we can keep the main operands in separate positive and negative forms, thereby solving the problem of direction of subtraction, and potentially simplifying the hardware unit. The algorithm was first published in [37], and in our paper we perform a quantitative comparison of multiple variants implemented in FPGA. Our work has been cited in 4 other publications at this time.

Comparing Subtraction-Free and Traditional AMI

Jiří Buček, Róbert Lórencz
Department of Computer Science and Engineering
Faculty of Electrical Engineering, Czech Technical University
Karlovo nám. 13, 121 35 Praha 2, Czech Republic
Email: bucekj@fel.cvut.cz, lorencz@fel.cvut.cz

***Abstract*— This paper presents FPGA implementations of traditional Almost Montgomery Inverse and Subtraction-free Almost Montgomery Inverse and compares their space and time properties. The subtraction-free algorithm with its hardware architecture overcomes the disadvantages of currently known methods (e.g. [2]). The ">" or "<" tests that require either extra clock cycles or extra chip area are completely eliminated.**

I. Introduction

Calculation of the modular inverse forms a part of various cryptographic algorithms, such as the decipherment of the RSA algorithm [10], digital signature systems [9], and so on. It is especially important in computing point operations on elliptic curves [8], or in accelerating the modular exponentiation operation using the so-called addition-subtraction chains [1]. Many cryptographic applications require an efficient hardware implementation of the modular inverse due to its time complexity and also for security reasons.

The majority of algorithms for computing the modular inverse are derived from the Extended Euclidean Algorithm [5]. The modular arithmetic operations are often performed in a Galois Field GF(p), where p is prime. An efficient "left-shifting" algorithm for computing the modular inverse in GF(p) is described in [6]. This algorithm avoids the "less-than/greater-than" tests that are in GF(p) subject to carry propagation delays. The algorithms discussed in this paper assume GF(p) as well.

One class of modular inversion algorithms computes the Montgomery Modular Inverse (MMI, [4]), which is defined as $MMI(a) \equiv a^{-1}2^{2n} \pmod p$, where $n = \lceil log_2 p \rceil$. MMI is computed in two steps. The first step computes the Almost Montgomery Inverse (AMI), $AMI(a) \equiv a^{-1}2^k$ $(\bmod\ p)$, where $k \in [n, 2n]$. The second step adjusts the result, multiplying it by 2^{2n-k}. In this paper, we focus only on the first step, thus computing AMI.

II. Subtraction-Free Almost Montgomery Inverse

This paper presents the first implementation of the Subtraction-Free AMI algorithm [7], which is a new variant of AMI published in [4].

The traditional algorithm for AMI uses the test $(u > v)$, which involves subtraction, and the sign of the difference determines the test result. If the condition is true, the result of this subtraction $(u - v)$ can be used in further processing (dividing by two, assigning into u). However, if the condition is false, result of the subtraction is negative and we need to compute the opposite value. When implementing in hardware, this requirement may present a significant overhead, since we must compute the difference $(v - u)$ or negate the previous result. A variant of the traditional AMI is given in Algorithm 1.

Algorithm 1, AMI with Subtractions

Input: $a \in [1, p-1]$ and $p > 2$ is prime
Output: $o \in [1, p-1]$ and k, where
$o = a^{-1}2^k \bmod p$ and $n - 1 \le k < 2n$

```
1.  u ← p, v ← a, r ← 0, s ← 1, k ← 0
2.  while(1)
3.      if (u_LSB == 0) then
4.          u ← u/2, s ← 2s
5.      else if (v_LSB == 0)
6.          v ← v/2, r ← 2r
7.      else
8.          x = u − v, y = r + s
9.          if (x == 0) then return o ← s and k
10.         if (CARRY(x) == 1) then
11.             u ← x/2, r ← y, s ← 2s
12.         else
13.             v ← (x = (v − u))/2, s ← y, r ← 2r
14.     k ← k + 1
```

Algorithm 2, Subtraction-Free AMI

Input: $a \in [1, p-1]$ and $p > 2$ is prime
Output: $o \in [1, p-1]$ and k, where
$o = a^{-1}2^k \bmod p$ and $n - 1 \le k < 2n$

```
1.  u ← (−p), v ← a, r ← 0, s ← 1, k ← 0
2.  do
3.      if (u_LSB == 0) then
4.          u ← u/2, s ← 2s
5.      else if (v_LSB == 0)
6.          v ← v/2, r ← 2r
7.      else
8.          x = u + v, y = r + s
9.          if (CARRY(x) == 0) then
10.             u ← x/2, r ← y, s ← 2s
11.         else
12.             v ← x/2, s ← y, r ← 2r
13.     k ← k + 1
14  while (x ≠ 0)
15. return o ← s and k
```

Algorithm 2 computes AMI completely without subtractions, thus avoids the overhead of computing both $(u-v)$ and $(v-u)$. To achieve the same result as in Algorithm 1, it computes $(u+v)$, where one of the addends must be negative. By keeping u always negative and v always positive, we can compute an equivalent of the differences in the original algorithm, without subtraction.

It can be shown that in every iteration of the AMI calculation, same values appear in v, r, s in Algorithm 2 as in Algorithm 1, while opposite values appear in u [7].

The negative values in u are represented using two's complement code. This allows us to use normal binary adders. Moreover, the information about the sign of u does not need to be stored explicitly, because u cannot be positive (and v cannot be negative). Therefore, we do not need any extra bits for storing the negative values, compared to the number of bits required to represent the always positive values in Algorithm 1.

By avoiding the $(u > v)$ test, our approach either saves chip area (compared to [2]; instead of two subtractors, one needs a single adder) or performs faster (compared to [3]; no extra clock cycles are needed). Furthermore, there is no need for multi-function arithmetic units (adders/subtractors).

There are other, more complex architectures for AMI and MMI, which deal with inversion of very long numbers (thousands of bits), thus focus on scalability [11] or use carry-free arithmetics [12]. Our implementations focus on shorter numbers (hundreds of bits), which are suitable for elliptic curves cryptography, and therefore we focus mainly on small and fast data paths and controllers.

A. Time Complexity

The time complexity of Algorithm 1 and 2 is determined by the number of operations in every iteration, which correspond to the iterations of the binary GCD algorithm. Knuth mentions in [5] that, for binary $\gcd(u, v)$, where $u, v \in [1, 2^n]$, the number of iterations and shifts $k \approx 2qn$, number of additions/subtractions $s \approx qn$, number of $(u > v)$ tests $t \approx (qn)/2$ (Algorithm 1 only), where $q \approx 0.70597$.

Assuming that all operations (additions, subtractions, shifts, tests) are performed in a single cycle and that we are able to add and shift in a single cycle, the speedup S of AMI calculation using Algorithm 2 compared to Algorithm 1 is: $S \approx \frac{k+t}{k} = 1 + \frac{(qn)/2}{2qn} = 1.25$. If additions, subtractions and $(u > v)$ tests take more clock cycles than shifts because of carry propagation delay, which would be true for large n $(n > 512)$, then the speedup is

$$S(f(n)) \approx \frac{k+f(n)s+t(1+f(n))}{k+f(n)s} = 1 + \frac{(1+f(n))/2}{2+f(n)}, \quad (1)$$

where $f(n)$ is a real function of a positive integer n such that $0 \leq f(n) \leq f(n+1)$ for all n. The function $f(n)$ represents additional clock cycles due to carry chain propagation delay. Using equation (1) with $f(n) = 0$ for some small n we obtain $S \approx 1.25$ as shown above. If $n \to \infty$, hence $f(n) \to \infty$, we obtain the maximum theoretical speedup

$$S_{max} \approx \lim_{f(n)\to\infty} S(f(n)) = 1.5,$$

which can be achieved with the proposed Algorithm 2 compared to Algorithm 1.

This of course assumes that operations with r and s are performed in parallel with operations performed with u, v as specified in Algorithm 1 and 2.

Unlike Algorithm 1, Algorithm 2 needs the negative value of p. It can be pre-calculated once for use in AMI as well as in other operations that require reductions modulo p. For example, $-p$ is actually more useful than p even for subsequent AMI to MMI conversion. The AMI only needs to be shifted certain number of times, subtracting p (that is, adding $-p$) whenever necessary to keep the result within $[1, p-1]$. Again, only an adder is needed.

III. AMI Implementations with Subtractors

When implementing AMI in hardware, we can divide the data path into the master part that computes $\gcd(u, v)$, and the slave part that computes the inverse (uses r, s). We have studied three different hardware units for computing AMI, which differ in the implementation of the master part of the data path, and in the corresponding controller.

These include AMI with one subtractor and AMI with two subtractors, which implement Algorithm 1, and Subtract-free AMI, which implements Algorithm 2. The slave part of the data path is always the same, it contains one adder for computing $(r+s)$.

Variants of implementation of Algorithm 1 involve computations of both $(u-v)$ and $(v-u)$. There are two possible approaches to this requirement – extra time or extra space (chip area). The first approach leads to AMI with one subtractor, which uses an extra clock cycle to compute the opposite difference on the same subtractor as the first difference. The second approach, AMI with two subtractors, uses the two subtractors to compute both differences concurrently.

A. AMI with One Subtractor

This implementation uses the time approach. The master part contains one subtractor to compute $x = (u-v)$. The output x of the subtractor is then used for the test $(u > v)$, and if the condition is true, x is used in subsequent computation. If the test condition is false (x is negative), the same subtractor is used again in the next clock cycle to compute the opposite difference $x = (v-u)$.

By using only one subtractor in the master part, we save area at the expense of extra clock cycles each time the test fails. However, some area overhead still applies – we have to connect multiplexers to the inputs of the subtractor. Hence the area saved by using only one subtractor depends on the particular technology platform, namely the area sizes of subtractors and multiplexers.

The number of clock cycles needed to compute AMI is greater than the number of iterations of the main loop of Algorithm 1

B. AMI with Two Subtractors

This implementation uses the space approach. There are two subtractors in the master part, therefore both differences, $(u - v)$ and $(v - u)$, are always available. The area is larger due to the additional subtractor, but there are no additional clock cycles, therefore this implementation is faster than AMI with one subtractor. The number of clock cycles needed to compute the inverse corresponds to the number of iterations of the algorithm main loop.

IV. Subtraction-Free AMI Implementation

The subtraction-free AMI implementation contains one adder in the master part, and one adder in the slave part. The result of addition $x = (u + v)$ is always divided by 2 (shifted right), and written into either u or v; the destination register determined by the sign of the result – this information is contained in carry.

The resulting architecture computes AMI in the same number of cycles as AMI architecture with two subtractors, but it is considerably smaller (contains 1 less adder). It is even possible to be slightly faster due to more simple logic (simpler multiplexers, no need to switch between $(u-v)$ and $(v-u)$).

The subtraction-free architecture is faster than AMI architecture with one subtractor while being approximately equal in area occupation. It is even possible to be both smaller and faster because it lacks the extra multiplexers.

V. Results

We have implemented in VHDL three different architectures for computing AMI: Subtraction-free AMI (SF-AMI, Section IV), AMI with one subtractor (1-SUB-AMI, Section III-A), and AMI with two subtractors (2-SUB-AMI, Section III-B). The target platform is a FPGA device Xilinx Virtex2 3000.

We have synthesized the generic designs using bit lengths of $n = 64, 128, 162$ and 256 bits, the results are presented in Table I. Area occupation is given in slices, the numbers are gathered from the map report of the Xilinx toolchain. The minimum clock period is gathered from the post place and route static timing report.

An important metric is the Time×area product (Table II), which reflects both the minimum time needed for the computation of AMI and the chip area occupied by the unit. Therefore, the numbers for AMI with one subtractor also reflect the average slowdown of 1.25 caused by the extra clock cycles needed when an opposite subtraction must be performed.

TABLE I
Area occupation (slices), minimum clock period (ns)

	SF-AMI		1-SUB-AMI		2-SUB-AMI	
n (bits)	area (slices)	period (ns)	area (slices)	period (ns)	area (slices)	period (ns)
64	347	15.328	411	14.631	393	14.855
128	638	14.369	801	15.189	710	17.777
162	917	19.660	955	19.709	1044	19.380
256	1272	24.591	1269	24.542	1491	24.610

TABLE II
Time×area product, scaled (slices*ns/1000)

n (bits)	SF-AMI	1-SUB-AMI	2-SUB-AMI
64	5.3	7.5	5.8
128	9.2	15.2	12.6
162	18.0	23.5	20.2
256	31.3	38.9	36.7

VI. Conclusion and Future Work

We have implemented the recently published Subtraction-free Almost Montgomery Inverse algorithm in FPGA and compared it to two different architectures for the traditional Almost Montgomery Inverse algorithm with subtractions.

Implementation results show that the Subtraction-free AMI algorithm [7] is suitable for hardware implementation and its implementation is equally fast as the implementation of AMI with two subtractors, yet about 13–17% smaller in area.

The Subtraction-free AMI implementation is equally small as the implementation of AMI with one subtractor, yet it is about 25% faster. We have observed that for smaller bit lengths, the Subtraction-free AMI implementation is both smaller and faster.

Our experiments with FPGA implementations were influenced by the fact that FPGAs contain dedicated hardware structures for carry chains (adders and subtractors). Our future work will focus on implementing the Subtraction-free algorithm in ASIC. Adding and subtracting numbers to about 256 bits is implementable using conventional binary adders and these word lengths are suitable for elliptic cryptography, finding its application, among others, in smart cards.

References

[1] Ö. Eğecioğlu, Ç. K. Koç, Exponentiation Using Canonical recoding, Theoretical Computer Science 129 (2), 1994, pp. 407–717.

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

[3] J. Hlaváč, R. Lórencz, Ordinary Modular Inverse Using AMI – Hardware Implementation, Proc. IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems–DDECS'03, Poznań, Poland, 2003, pp. 309–310.

[4] B. S. Kaliski Jr., The Montgomery Inverse and Its Application, IEEE Transaction on Computers 44 (8), 1995, pp. 1064–1065.

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

[6] R. Lórencz, New Algorithm for Classical Modular Inverse, Cryptographic Hardware and Embeded Systems - CHES 2002, Springer-Verlag LNCS 2523, 2003, pp. 57–70.

[7] R. Lórencz, J. Hlaváč, Subtraction-free Almost Montgomery Inverse Algorithm, Information Processing Letters 94 (1), 2005, pp. 11–14.

[8] A. J. Menezes, Elliptic curve Public Key Cryptosystem, Kluwer Academic Publishers, Boston, MA, 1993.

[9] Nat'l Inst. of Standards and Technology (NIST), FIPS Publication 186: Digital Signature Standard, 1994.

[10] J.-J. Quisquarter, C. Couvreur, Fast Decipherment Algorithm for RSA Public-key Cryptosystem, Electronics Letters 18 (21), 1982, 905–907.

[11] E. Savaş, M. Naseer, A. A-A. Gutub, Ç. K. Koç, Efficient unified Montgomery inversion with multibit shifting, IEE Proceedings – Computers and Digital Techniques 152 (4), 2005, pp. 489–498

[12] E. Savaş, A Carry-Free Architecture for Montgomery Inversion, IEEE Transactions on Computers 54 (12), 2005, pp. 1508–1519

5.3 RP3 – Dedicated Hardware Implementation of a Linear Congruence Solver in FPGA

This paper was published in *IEEE International Conference on Electronics, Circuits and Systems (ICECS) 2012* [A.4]. The third paper examines the FPGA hardware implementation of a solver of systems of linear congruences, the main part of the linear solver.

Main topic of this paper is the architecture of the arithmetic units and memory suitable or FPGA, and the evaluation of the resulting hardware implementation. Particular interest lies in the memory architecture, since it forms a bottleneck in the computation, and it must support the special operations needed in solving linear systems, namely pivoting and result reordering.

The work is based on [1] and [2] and is further used and extended in our subsequent relevant papers **RP4-6**.

Dedicated Hardware Implementation of a Linear Congruence Solver in FPGA

Jiří Buček, Pavel Kubalík, Róbert Lórencz, and Tomáš Zahradnický
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9, 160 00 Prague, Czech Republic
Email: { bucekj | xkubalik | lorencz | zahradt }@fit.cvut.cz

***Abstract*— The residual processor is a dedicated hardware for solving sets of linear congruences. It is a part of the modular system for solving sets of linear equations without rounding errors using Residue Number System. We present a new FPGA implementation of the residual processor, focusing mainly on the memory unit that forms a bottleneck of the calculation, and therefore determines the effectivity of the system. FPGA has been chosen, as it allows us to optimally implement the designed architecture depending on the size of the problem. The proposed memory architecture of the modular system is implemented using the internal FPGA block RAM. Our goal is to determine the maximum matrix dimension fitting directly into the FPGA, and achieved speed as a function of the dimension. Experimental results are obtained for the Xilinx Virtex 6 family.**

I. Introduction

Residue Number System (RNS), despite being known for a long time, is becoming a hardware attractive arithmetic today, not only because it permits us to represent long integers as independent combinations of small integers based on the Chinese Remainder Theorem, but also because it requires a simple arithmetic unit. These properties offer natural parallelism, lead to simpler hardware, and reduce chip size when compared to a traditional floating point unit implemented in hardware.

RNS is used in areas of digital image processing [1][2], digital signal processing [3][4][5], and in public-key [6] and elliptic curve [7][8] cryptography. RNS is also used to simulate multiple precision arithmetic and for error-free solution of linear systems [9][10]. Error-free solution of linear systems is often needed in case of large, dense and ill-conditioned systems, where rounding errors can lead to long run times due to stability problems, or even hinder the solution completely.

Performing error-free solution of linear systems on regular CPUs has large time (and area) complexity. The CPU architecture is usually not optimized for the algorithms and operations needed (parallelism with respect to multiple modules, modular arithmetic operations etc).

Papers [11] and [12] design a hardware RNS linear equation solver — Modular System (MS) — whose implementation was very difficult at that time. With current technologies, it is possible to implement the system, and especially FPGA technologies offer a straightforward implementation with reconfiguration possibilities based on the cardinality of the problem and optimize for time and area.

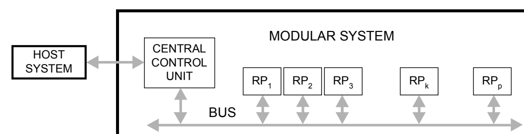


Fig. 1. Architecture of the Modular System [11]

However, for an efficient FPGA implementation, several parts of the system must be redesigned to use resources found in modern FPGAs. This is true especially for the memory architecture, which in [12] used asynchronous logic and custom memory elements. We present a new memory architecture using standard RAM blocks found in most recent FPGAs. We also redesigned the addressing and pivoting logic to support efficient implementation of the elimination algorithm used.

After a brief introduction of the architecture of the MS for solution of sets of linear equations (SLE)s $\mathbf{Ax} = \mathbf{b}$, the paper focuses on the memory architecture of the residual processors (RP)s inside the MS. Next, there follow FPGA implementation results for various problem sizes, their analyses, and evaluations. Finally, the paper is concluded with the properties of the FPGA residual processor implementation.

II. Architecture of the Modular System

Paper [12] describes the method, the algorithm, and the corresponding parallel hardware architecture of the MS (Fig. 1).

It should be noted that evaluation in each modulus is performed independently of the others and that the addition is carry-free, subtraction borrow-free across the individual moduli, and therefore the computation can occur safely in parallel. Once the computation is done, the result is transformed back into the rational number set either with the Chinese Remainder Theorem or the Mixed Radix Conversion.

The error-free solution of an SLE with operations performed in residue arithmetic is implemented in this special MS. The MS typically has a parallel SIMD architecture, and consists of a control unit and several processing units – (RP)s denoted as RP_1, RP_2, …, RP_p interconnected with a BUS (see Fig. 1).

III. Residual Processor Architecture

The architecture of the residual processor RP_k is depicted in Fig. 2 consisting of Memory, Arithmetic Units AU_2, AU_3, …, AU_{n+2} and the Control unit.

978-1-4673-1260-8/12/$31.00 ©2012 IEEE 689

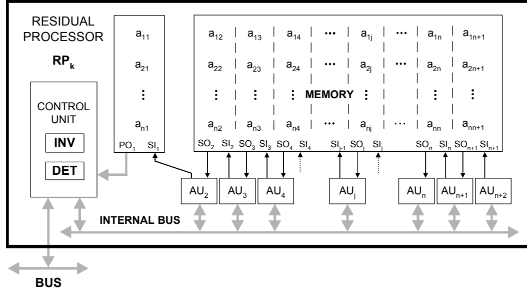


Fig. 2. Architecture of the Residual Processor [11]

The memory contains residues of matrix $\mathbf{A}$ and vector $\mathbf{x}$ elements. The storage of values of a row of the matrix from AU registers is performed bitwise via Serial Inputs SI_1, SI_2, $\ldots$, SI_{n+1}. Loading of values of rows of Memory to AUs is done via Serial Outputs SO_2, SO_3, $\ldots$, SO_{n+1}. The bits of element values of the first matrix column are read by the Control unit via the parallel bus PO_1. All AUs and the Control unit are interconnected via Internal Data Buses $\mathrm{IDB}_{\mathrm{in}}$ and $\mathrm{IDB}_{\mathrm{out}}$. The above RP_k architecture can solve systems of linear congruences (SLC)s $\mathbf{A}\mathbf{x}_k \equiv \mathbf{b} \pmod{m_k}$. RPs together with the Control unit of the MS also support all conversion operations from integer to RNS and vice versa. The INV and DET units compute the modular multiplicative inverse and the determinant of $\mathbf{A}$, respectively.

All SLCs in MS are solved with the Gauss-Jordan elimination with Rutishauser modification [12] (GJR), which is especially suitable for hardware implementation. The elimination process in RNS is specific in a way that it has to perform a so called "nonzero residue pivoting" that was introduced in [11]. Pivoting and massive data access constitute a bottleneck, and therefore the memory architecture design is critical and is dealt with in the next section.

The Rutishauser modification of Gauss-Jordan elimination implies that the column data is shifted by one column to the left during each elimination step. The shift is accomplished by the AUs and the memory interconnection design in Fig. 2. In addition, the first column of the SLC matrix contains values of the elements intensively used during the elimination process and for this reason the output from the first column needs to be parallel (these values are used in the INV and DET units). The values in the first column determine the first multiplication operand in the entire row being processed, both in pivot elimination and row reduction. The other columns a_{i2} to a_{in+1} inclusive are used as the second multiplication operand, and also for addition operations. Assuming serial-parallel (shift-add) multiplication, we need to read individual bits of these values, thus requiring serial access only.

A. New Memory Architecture and Pivoting

The elimination process requires nonzero residue pivoting. The pivot column is always the first column of the matrix, and all nonzero values are equally acceptable as pivots. Search for a pivot is done sequentially; however, this search can be easily

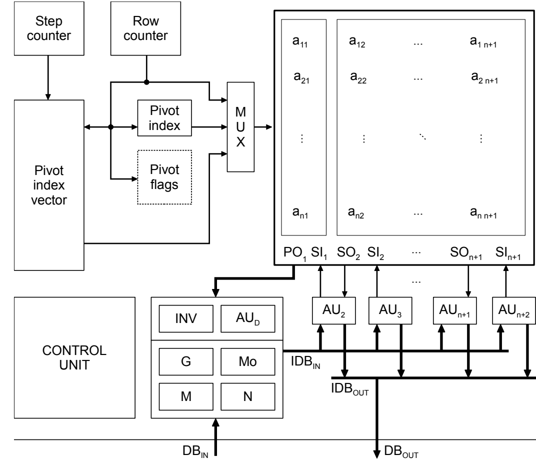


Fig. 3. New architecture of the Residual Processor including new memory architecture

performed concurrently with write operations to the memory. The search is performed while the matrix is loaded or updated during computation. In most cases, the pivot is passed to the inversion unit (INV) long before the inverse is needed. In order for a value to be accepted as a pivot, i) it must be nonzero, ii) the row has not contained a pivot yet, and iii) no pivot has yet been found for this elimination step.

Once the pivot is found, its row index must be stored in a pivot index vector at the address of the current elimination step. The pivot row must be flagged in order to skip it during the pivot search performed in subsequent elimination steps. If no pivot was found, the matrix is singular in this modulus.

The elimination is performed by rows. The architecture must support addressing of the pivot row first; then sequentially reduce memory matrix rows, with an exception of the pivot row which must be skipped. The first value in each row must be read in parallel. This value is either the pivot, which is inverted, or a value from a different row, which is negated.

The remaining values in each row are read bit-serial (but all values concurrently) from the MSb first. This ensures the correct order for left-shift modular multiplication and addition, and follows from the design depicted at Fig. 3.

Upon completion of the elimination process, the solution vector appears in the first column. The order of its elements corresponds to the pivot row indices and may need to be reordered. The result is therefore read out in correct order by addressing through the pivot index vector.

Algorithm 1. Elimination algorithm including the pivoting instructions from Table I. Parameters: n is matrix dimension, e is word length.

1. $k = 0$, assuming pivot is found during loading of matrix
2. **while** $k < n$ **begin**
3. PSI
4. GETD

TABLE I
PIVOTING INSTRUCTIONS USED IN ALGORITHM 1.

Instruction	Description
PSI	Select pivot row
GETD	Read data row from memory
PUT	Write data to memory and test for pivot
PTS	Test and skip pivot row during reduction
PCRR	Reset row counter

5. multiplication of the pivot row with pivot^{-1} ($5e$ clocks)
6. PUT
7. PCRR
8. **repeat** $n-1$ **times begin**
9. PTS
10. GETD
11. reduction of other matrix rows using adjusted pivoting row ($5e$ clocks)
12. PUT
13. **end repeat**
14. **end while**

B. Arithmetic Units, Modular Inverse and Controller

The Arithmetic Units AU_2, AU_3, $\ldots$, AU_{n+2} and AU_D design closely follows the original design in [12]. The modular inversion unit INV was originally designed as a look-up table. However, for larger moduli, this table grows too large, and therefore we use a new inversion unit computing modular inverse with the left-shift modular inverse algorithm [13].

The controller contains a finite state machine using a memory-based transition and output functions. This allows flexibility with regard to modification and future extensions.

C. FPGA Implementation

The memory architecture as a critical part of the RP can be divided into two parts: the pivoting unit and common memories. The pivoting unit is always implemented in FPGA. Memory can be implemented internally using block RAM components, or externally e.g. by a DDR SDRAM. The main implementation differences are in their parameters such as memory capacity, throughput, and latency. On one hand, the internal implementation with FPGA memory has small capacity and low latency, while on the other hand the external memory provides large capacity but also a high latency.

We design an architecture with the internal memory. We can estimate the size of the largest matrix with respect to maximum size of the block RAM given by the FPGA chip type. Nonetheless, the maximum frequency of the implemented design cannot be easily estimated or calculated. Our tested memory architecture consists of two parts: i) the internal memory, and ii) a pivoting control logic to support addressing during the calculation in the RP. The design of our memory architecture is shown in Fig. 3.

The memory matrix consists of the first column a_{i1} and the remaining columns a_{i2} to $a_{i,n+1}$. All columns share a common address. During pivot search, the address is taken from the Row counter and if the pivot is found in the current row, the address is written into the Pivot index vector at the address of current elimination step, and the Pivot flag for the address of the current row is set. At the same time, the pivot address is stored in the Pivot index register for comparison during the next elimination step.

IV. EXPERIMENTAL RESULTS

All experiments were conducted on the residual processor architecture consisting of: data memory, Pivot index, Pivot flags, counters, arithmetic units, inversion unit, and control units. The tools used for simulation and implementation were selected with respect to the hardware programming language. The design was written in VHDL. The maximum matrix dimension n and the word length e and are configurable at synthesis time using generics. The actual matrix dimension, value of the modulus and matrix data are set at runtime.

The design was simulated, synthesized and implemented (mapped, placed and routed). The experiments were performed on one residual processor (single modulus), including the input data modulo reduction and matrix elimination. (Transformation into the rational numbers was not performed).

To verify the design, we added test units increasing testability and observability of the simulated design to verify the calculation. The test data were generated using Wolfram Mathematica and converted with a Python script. The simulation of the residual processor architecture ran within Mentor ModelSim. The simulation results and Mathematica results were compared by another script. The simulation was done for matrices up to $n = 100$, greater matrices were not simulated due to a high simulation time.

The implementation process started after simulation, when correctness matrix calculation was verified. We tested several different matrix dimensions, and always set the word length to 24 bits. The block RAM modules were inferred from a functional description by the synthesis tool. Each column (a_{i1} to $a_{i,n+1}$) is implemented as a block RAM module. The memory is not always used effectively, depending on the number of rows. Each memory module has the full capacity given by the width of its address bus, that is a power of two. Therefore the memory utilization increases by a step when the number of address bits changes. The number of arithmetic units was automatically generated by the selected matrix dimension.

The Xilinx FPGA platform was selected for all tests. We used Xilinx ISE to synthesize and implement our design to the FPGA. We selected the FPGA with the highest block RAM memory capacity in the Virtex 6 family, that is, the xc6vsx475t-2-ff1156. In order to get a good estimate of the best achievable timing, we set the "High effort" with "Continue on Impossible" options for the implementation part. Constraining the timing would achieve even better timing. We gathered the minimum period, logic and memory utilization from the implementation (post place and route) report files. The results of our experiments are shown in Table II.

TABLE II
IMPLEMENTATION RESULTS FOR THE FPGA RESIDUAL PROCESSOR ARCHITECTURE (FPGA IS XILINX XC6VSX475T).

	Area utilization				Time		
n	slices	%slices	BRAM	%BRAM	T_P[ns]	f_{clk}[MHz]	T_{elim} [ms]
100	4223	6%	101	10%	6	166	7.2
300	12194	16%	301	28%	7	143	74.8
500	19277	26%	501	47%	9	111	266.6
700	29394	40%	701	66%	11.4	88	658.2
900	38372	52%	901	85%	12.7	78	1216.6
1000	42368	57%	1001	94%	13	77	1537.1

The n column denotes the matrix dimension. The "slices" and "BRAM" columns are the number of used slices and used block RAMs also with the percentage of occupied FPGA resources (xc6vsx475t). The "T_P" and "f_{clk}" columns are minimum clock periods and maximum operation frequencies. The "T_{elim}" column shows the time in milliseconds to solve a set of linear congruences for one modulus depending on minimum clock period for the selected matrix dimension.

The elimination time (T_{elim}) assumes the data are already loaded and stored in memory and elimination process is in run. The load part takes only a small part of all time needed for solution of a set of linear congruences. For example, for matrix dimension $n = 100$, the load process takes only 12.7%, while for $n = 1000$ it takes only 11.7%.

The results show that our residual processor architecture allows for a maximum matrix size of approx. 1000 rows by 1001 columns with a word size of 24 bits in the chosen FPGA type. Even with the maximum tested matrix dimension of 1000, which uses more than 90% of the available block RAM, only approx. 60% of all available slices in FPGA are used.

The clock period increases with the increasing matrix dimension. Static time analysis shows that the main parts of the delay in the circuit are in addressing, control and inner data bus signals. The fanout of signals significantly increase when size of matrix increases. For comparison, on a CPU, solving a SLC of dimension 100, 500 and 1000 takes approximately 3 ms, 424 ms, and 3.37 s, respectively (Intel T9400 CPU at 2.53GHz, cache size 6144 KB, C language compiled with GCC). This shows that for $n = 1000$, our design is approx. 2 times faster. In case of future ASIC implementation of our design, we can expect even greater speedup (which is difficult to predict, but can reach 100 or more).

Further work will focus on the design of an external memory interface, which will be influenced by the limited FPGA input/output pins. The acquired results will be used to design the whole system for solving SLEs.

V. CONCLUSION

We have designed a Residual Processor (RP) architecture which solves a set of linear congruences. RP was designed with a focus on its effective implementation in FPGA for various problem sizes with a special attention to the memory architecture. The memory design is critical to the RP because of massive data access and pivoting. RPs are portions of a Modular System for solving sets of linear equations in RNS.

All important parts of the RP architecture, such as data memory, Pivot index, Pivot flags, counters, arithmetic units, inversion unit and control unit were implemented and tested in a Xilinx Virtex 6 FPGA with the largest RAM size. The results show that our RP architecture allows for a maximum matrix size of approx. 1000 rows by 1001 columns with a word size of 24 bits in the chosen FPGA type. The maximum tested matrix dimension of 1000 uses more than 90% of the available block RAM and approximately 60% of all available slices in the FPGA while being approx. 2 times faster than a CPU software implementation.

Future work will focus on a new RP architecture with external memory and limited numbers of AUs. Also bitwise communication between internal memory and AUs will be studied. Next, we will implement the RP in ASIC and evaluate its performance.

ACKNOWLEDGMENT

This research was supported by the Czech Science Foundation project no. P103/12/2377.

REFERENCES

[1] D. Taleshmekaeil and A. Mousavi, "The use of residue number system for improving the digital image processing," in *Signal Processing (ICSP), 2010 IEEE 10th International Conference on*, oct. 2010, pp. 775–780.

[2] T. Toivonen and J. Heikkila, "Video filtering with fermat number theoretic transforms using residue number system," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 16, no. 1, pp. 92–101, jan. 2006.

[3] G. Cardarilli, A. Nannarelli, and M. Re, "Residue number system for low-power DSP applications," in *Signals, Systems and Computers, 2007. ACSSC 2007. Conference Record of the Forty-First Asilomar Conference on*, nov. 2007, pp. 1412–1416.

[4] R. Chaves and L. Sousa, "RDSP: a RISC DSP based on residue number system," in *Digital System Design, 2003. Proceedings. Euromicro Symposium on*, sept. 2003, pp. 128 – 135.

[5] A. Mirshekari and M. Mosleh, "Hardware implementation of a fast FIR filter with residue number system," in *Industrial Mechatronics and Automation (ICIMA), 2010 2nd International Conference on*, vol. 2, may 2010, pp. 312 –315.

[6] J.-C. Bajard and L. Imbert, "A full RNS implementation of RSA," *Computers, IEEE Transactions on*, vol. 53, no. 6, pp. 769 –774, june 2004.

[7] D. Schinianakis, A. Kakarountas, and T. Stouraitis, "A new approach to elliptic curve cryptography: an RNS architecture," in *Electrotechnical Conference, 2006. MELECON 2006. IEEE Mediterranean*, may 2006, pp. 1241–1245.

[8] T. Güneysu and C. Paar, "Ultra high performance ECC over NIST primes on commercial FPGAs," in *Proceedings of the 10th international workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 62–78. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85053-3-5

[9] R. T. Gregory and E. V. Krishnamurthy, *Methods and Application of Error-free Computation*. Springer Verlag, 1984.

[10] D. M. Young and R. T. Gregory, *A Survey of Numerical Mathematics*, Addison-Wesley Series in Mathematics ed. Addison-Wesley Publishing Company, Inc., 1973, vol. 2.

[11] R. Lórencz and M. Morháč, "A modular system for solving linear equations exactly, i. architecture and numerical algorithms," *Computers and Artificial Intelligence*, vol. 11, no. 4, pp. 351–361, 1992.

[12] M. Morháč and R. Lórencz, "A modular system for solving linear equations exactly, ii. hardware realization," *Computers and Artificial Intelligence*, vol. 11, no. 5, pp. 497–507, 1992.

[13] R. Lórencz, "New algorithm for classical modular inverse," in *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES '02. London, UK, UK: Springer-Verlag, 2003, pp. 57–70.

5.4 RP4 – Comparison of FPGA and ASIC Implementation of a Linear Congruence Solver

This paper was published in *Euromicro Conference on Digital System Design (DSD 2013)* [A.5]. In this paper, we adapt the architecture of the solver of systems of linear congruences to ASIC technology. Memory architecture was adapted to the memory compilers available with the ASIC technology library. The results are compared with FPGA. The results of this paper are used in our subsequent relevant papers **RP5-6** in order to examine the possibilities of a hardware system for solving systems of linear equations.

2013 16th Euromicro Conference on Digital System Design

Comparison of FPGA and ASIC Implementation of a Linear Congruence Solver

Jiří Buček, Pavel Kubalík, Róbert Lórencz, Tomáš Zahradnický
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9, 160 00 Prague, Czech Republic
email: { bucekj | xkubalik | lorencz | zahradt }@fit.cvut.cz

***Abstract*—Residual processor (RP) is a dedicated hardware for solution of sets of linear congruences. RPs are parts of a larger modular system for error-free solution of linear equations in residue arithmetic. We present new FPGA and ASIC RP implementations, focusing mainly on their memory units being a bottleneck of the calculation and therefore determining the efficiency of the system. First, we choose an FPGA to easily test the functionality of our implementation, then we do the same in ASIC, and finally we compare both implementations together. The experimental FPGA results are obtained for Xilinx Virtex 6, while the ASIC results are obtained from Synopsys tools with a 130 nm standard cell library. Results also present a maximum matrix dimension fitting directly into the FPGA and achieved speed as a function of the dimension.**

***Keywords*-system of linear equations; residue number system; error-free computation; FPGA; ASIC**

I. Introduction

Solving systems of linear equations is one of the most frequent tasks in scientific computation. Traditionally, solution of such systems is carried out in floating point arithmetic bringing its associated rounding errors. Although there are algorithms minimizing the impact of rounding errors upon the solution, in some cases this does not need to be enough. An error-free solution of linear systems is often needed in case of large, dense, and possibly ill-conditioned systems, where needless rounding errors can result in longer run times, or even swamp the solution completely. One of the methods to go around rounding errors is to perform an error-free computation by means of residue arithmetic — Residue Number System (RNS).

RNS permits us to represent long integers as independent combinations of small integers based on the Chinese Remainder Theorem. It requires a simple arithmetic unit without any rounding and normalization logic that would be needed for floating point calculation. These properties offer natural parallelism, lead to a simpler hardware, and reduce chip size when compared to a traditional floating point unit implemented in hardware.

RNS is used in areas of digital image processing [1], digital signal processing [2], and in public-key [3] and elliptic curve [4] cryptography. RNS is also used to simulate multiple precision arithmetic and for error-free solution of linear systems [5].

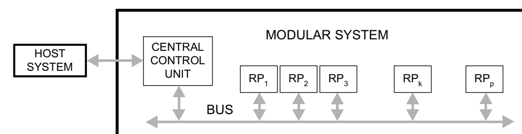


Figure 1. Architecture of the Modular System [7]

Our previous work [6] presents an FPGA implementation of a residual processor (RP) being a dedicated hardware for solution of sets of linear congruences. In this paper, we implement RP in ASIC and compare the achieved time and area complexity to its FPGS implementation.

Papers [7] and [8] design a hardware RNS linear equation solver — Modular System (MS) — whose implementation was very difficult at that time. With current technologies, it is possible to implement the system, either using FPGAs offering a straightforward implementation with reconfiguration possibilities, or in ASIC to achieve higher speeds and possibly a lower price, when produced in larger quantities.

After an extensive redesign in [6] to use block RAM cells found in most recent FPGAs, we implemented the architecture in ASIC using a 130 nm technology with standard cells and compiled memory modules. We also redesigned the addressing and pivoting logic to support efficient implementation of the elimination algorithm used.

After a brief introduction of the MS architecture performing solution of sets of linear equations (SLE)s $\mathbf{Ax} = \mathbf{b}$, the paper focuses on the memory architecture of RPs inside MS. Next, there follow FPGA and ASIC implementation results for various problem sizes, their analyses, and evaluations. Finally, the paper is concluded with FPGA and ASIC RP implementation properties.

II. Architecture of the Modular System

Paper [8] describes the method, the algorithm, and the corresponding parallel hardware architecture of the MS (Fig. 1). Evaluation in each modulus is performed independently of the others with the addition carry-free, subtraction borrow-free across the individual moduli, and thus the computation can occur safely in parallel. Once the computation is done, the result is transformed back into the rational number set either with the Chinese Remainder Theorem or the Mixed Radix Conversion.

'/13 $26.00 © 2013 IEEE
DOI 10.1109/DSD.2013.125

CPS Conference Publishing Services

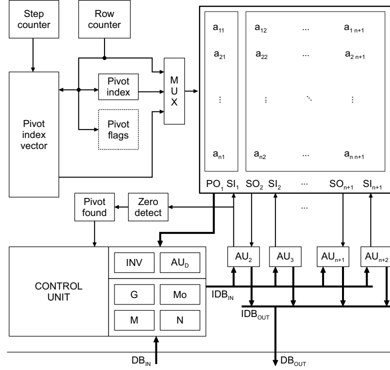


Figure 2. Architecture of the Residual Processor

The error-free solution of an SLE with operations performed in residue arithmetic is implemented in this special MS. The MS typically has a parallel SIMD architecture consisting of a control unit and several processing units — RPs denoted as RP_1, RP_2, ..., RP_p interconnected with a BUS (see Fig. 1).

III. Residual Processor Architecture

The architecture of the residual processor RP_k is depicted at Fig. 2 and consists of Memory, Arithmetic Units AU_2, AU_3, ..., AU_{n+2} and the Control unit.

The memory contains residues of matrix $\mathbf{A}$ and vector $\mathbf{x}$ elements. The storage of values of a row of the matrix from AU registers is performed bitwise via Serial Inputs SI_1, SI_2, ..., SI_{n+1}. Loading of values of rows of Memory to AUs is done via Serial Outputs SO_2, SO_3, ..., SO_{n+1}. The bits of element values of the first matrix column are read by the Control unit via the parallel bus PO_1. All AUs and the Control unit are interconnected via Internal Data Buses IDB_{in} and IDB_{out}. The above RP_k architecture can solve systems of linear congruences (SLC)s $\mathbf{A}\mathbf{x}_k \equiv \mathbf{b} \pmod{m_k}$. RPs together with the Control unit of the MS also support all conversion operations from integer to RNS and vice versa. The INV and DET units compute the modular multiplicative inverse and the determinant of $\mathbf{A}$, respectively.

All SLCs in MS are solved with the Gauss-Jordan elimination with Rutishauser modification [8] (GJR), which is especially suitable for hardware implementation. The elimination process in RNS is specific in a way that it has to perform a so called "nonzero residue pivoting" that was introduced in [7]. Pivoting and massive data access constitute a bottleneck, and therefore the memory architecture design is critical and is dealt with in the next section.

The Rutishauser modification of Gauss-Jordan elimination implies that the column data is shifted by one column to the left during each elimination step. The shift is accomplished by the AUs and the memory interconnection design in Fig. 2. In addition, the first column of the SLC matrix contains values of the elements intensively used during the elimination process and for this reason the output from the first column needs to be parallel (these values are used in the INV and DET units). The values in the first column determine the first multiplication operand in the entire row being processed, both in pivot elimination and row reduction. The other columns a_{i2} to a_{in+1} inclusive are used as the second multiplication operand, and also for addition operations. Assuming serial-parallel (shift-add) multiplication, we need to read individual bits of these values, thus requiring serial access only.

A. Memory Architecture in FPGA and ASIC

In order for a value to be accepted as a pivot, i) it must be nonzero, ii) the row has not contained a pivot yet, and iii) no pivot has yet been found for this elimination step. The pivot is always in the first column of the matrix. Search for a pivot is done sequentially; however, this search can be easily performed concurrently with write operations to the memory. The search is performed while the matrix is loaded or updated during computation. In most cases, the pivot is passed to the inversion unit (INV) long before its inverse is needed.

Once the pivot is found, its row index must be stored in a pivot index vector at the address of the current elimination step. The pivot row must be flagged in order to skip it during the pivot search performed in subsequent elimination steps. If no pivot was found, the matrix is singular in this modulus.

The elimination is performed by rows. The architecture must support addressing of the pivot row first; then sequentially reduce memory matrix rows, with an exception of the pivot row which must be skipped. The first value in each row must be read in parallel. This value is either the pivot, which is inverted, or a value from a different row, which is negated.

The remaining values in each row are read bit-serial (but all values concurrently) from the MSb first. This ensures the correct order for left-shift modular multiplication and addition, and follows from the design depicted at Fig. 2.

Upon completion of the elimination process, the solution vector appears in the first column. The order of its elements corresponds to the pivot row indices and may need to be reordered. The result is therefore read out in correct order by addressing through the pivot index vector.

B. Arithmetic Units, Modular Inverse and Controller

The Arithmetic Units AU_2, AU_3, ..., AU_{n+2} and AU_D design is modified from the original circuit in [8] by using strictly synchronous design. It supports computation of modulo operation on multi-word inputs, which is used when loading a new matrix into the modular system. During elimination, it computes modular multiplication and addition operations. Multiplication operations are performed using a shift-add algorithm with interleaved modulus subtraction.

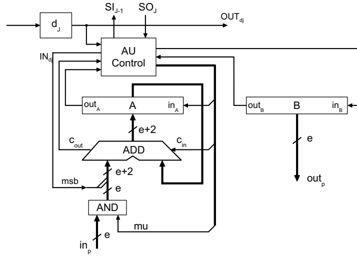


Figure 3. Arithmetic Unit architecture

The block diagram of the arithmetic unit is given in Fig. 3. The main part of the unit is the adder that performs all addition and subtraction operations including elementary additions during multiplication. The intermediate results are stored in the A register. The B register serves as a temporary storage for the values of the multiplied pivot row to be used during row reduction.

For computing the modular inverse in the INV unit, the left-shift modular inverse algorithm [9] is used.

The controller contains a finite state machine using a memory-based transition and output functions. This allows flexibility with regard to modification and future extensions.

C. FPGA and ASIC Implementation

The memory architecture as a critical part of the RP can be divided into two parts: the pivoting unit and data memory containing the matrix. The pivoting unit is always implemented using on-chip logic and memory blocks in FPGA or ASIC. The data memory can be implemented internally using block RAM components, or externally e.g. by a DDR SDRAM. The main implementation differences are in their parameters such as memory capacity, throughput, and latency. On one hand, the internal implementation with FPGA memory has small capacity and low latency, while on the other hand the external memory provides large capacity but also a high latency.

In order to compare the implementations in FPGA and ASIC, we used internal memory in both cases. For a given FPGA type, we can estimate the size of the largest matrix that fits on chip according to maximum size of the block RAM. In ASIC, we do not have such hard limits, but we can observe the occupied chip area as a function of the internal memory capacity. In both cases, we analyze the maximum frequency of the design after implementation (or, in the case of ASIC, after synthesis). Our tested memory architecture consists of two parts: i) the internal memory, and ii) a pivoting control logic to support addressing during the calculation in the RP. The design of our memory architecture is shown in Fig. 2.

The memory matrix consists of the first column a_{i1} and the remaining columns a_{i2} to $a_{i,n+1}$. All columns share a common address. During pivot search, the address is taken from the Row counter and if the pivot is found in the current row, the address is written into the Pivot index vector at the address of current elimination step, and the Pivot flag for the address of the current row is set. At the same time, the pivot address is stored in the Pivot index register for comparison during the next elimination step.

Table I
IMPLEMENTATION RESULTS FOR THE FPGA RESIDUAL PROCESSOR ARCHITECTURE (FPGA IS XILINX XC6VSX475T).

n	Area Utilization					Time
	BRAM	Mem [sl]	Au [sl]	Cntl [sl]	**All [sl]**	**All [MHz]**
100	103	480	2889	77	**4223**	**166**
300	303	1269	8920	87	**12117**	**142**
700	701	2821	21438	73	**29560**	**79**
1000	1001	4068	30095	73	**42368**	**76**

IV. EXPERIMENTAL RESULTS

All experiments were conducted on an RP architecture consisting of data memory, Pivot index, Pivot flags, counters, arithmetic units, an inversion unit, and control units. The final design was written in VHDL. The maximum matrix dimension n and the word length e are configurable at synthesis time using generics, while matrix dimension n, modulus, and the matrix data are set at runtime.

The design was simulated, synthesized and implemented (only for FPGA) (mapped, placed, and routed). The experiments were performed on 1 RP (with a single modulus), including the input data modulo reduction and matrix elimination. A transformation into the rational numbers set was not performed. The experiments were performed separately for FPGA and ASIC.

After verification of a simulation correctness we started an implementation in FPGA. The Xilinx FPGA platform was selected for all FPGA tests in Xilinx ISE. We selected the FPGA with the highest block RAM memory capacity in the Virtex 6 family i.e xc6vsx475t-2-ff1156. The results of our experiments are shown in Table I.

The n column denotes the matrix dimension. In the area utilization part the "All" and "BRAM" columns are the number of slices and Block RAMs for whole residual processor implementation. The "Mem", "Au" and "Control" columns show the number of slices used only for the memory, AUs and for the controller. The "Time" column contains the maximum frequency obtained from the implementation.

For ASIC we performed only the synthesis. Since block RAMs were not in the ASIC library as standard cells, we generated them with a special memory generator tool that comes with the library.

Synopsys Design Compiler was used for all ASIC tests with a 130 nm technology library. The results of our experiments are shown in Table II. The table headings are similar to Table I, but area figures are given in mm^2. The maximum frequency is obtained from synthesis.

Table II
SYNTHESIS RESULTS FOR THE ASIC RESIDUAL PROCESSOR ARCHITECTURE (130 NM LIBRARY).

n	Area Utilization				Time
	Mem [mm^2]	Au [mm^2]	Control [sl]	**All [mm^2]**	**All [MHz]**
100	2,22	0,82	0,015	**3,08**	**380**
300	14,02	2,22	0,016	**16,57**	**348**
700	52,41	5,09	0,017	**58,16**	**309**
1000	93,53	7,34	0,017	**101,74**	**310**

The number of clock cycles needed for load and elimination process can be calculated by (1) and (2), where n is matrix size, with z bits per word and q input words.

$$\mathrm{load}(n,z,q) = ((((2(n+1))+(1+2+2z))q)+7)n+1, \quad (1)$$

$$\mathrm{elim}(n,z) = ((z+(4z-2))n+3)n+14. \quad (2)$$

To calculate the elimination time (T_{elim}) we assume, that the data are already loaded and stored in memory and elimination process is in run. The load part takes only a small part of all entire SLC solution time. For example, for an $n = 100$ matrix, the loading process takes only 12.7 %, while for $n = 1000$ it takes only 11.7 %. As we can calculate, the elimination time for a 100 column matrix is 7 ms for FPGA and 3 ms for ASIC. For a 1000 column matrix, the elimination time is 1552 ms for FPGA and 380 ms for ASIC.

When we run the same task on a CPU solving an SLC of dimensions 100, 500, and 1000, it takes approximately 3 ms, 424 ms, and 3.37 s, respectively calculated on Intel T9400 CPU running at 2.53 GHz with a 6144 KB cache. This shows that for $n = 1000$, our design is approximately 5 times faster.

The results show that our residual processor architecture allows for a maximum matrix size of approximately 1000 rows by 1001 columns with a word size of 24 bits in the chosen FPGA type. Even with the maximum tested matrix dimension of 1000, which uses more than 90 % of the available block RAM, only approximately 60 % of all FPGA available slices are used.

The clock period increases with an increasing matrix dimension. Static time analysis shows that the main parts of the delay in the circuit are in addressing, control and inner data bus signals. The fanout of signals significantly increases when the size of matrix increases.

V. Conclusion

We have designed a Residual Processor (RP) architecture performing a solution of a set of linear congruences. RP was designed with a focus on its effective implementation in FPGA and ASIC for various problem sizes with a special attention to the memory architecture.

All important parts of the RP architecture were implemented and tested in Xilinx Virtex 6 FPGA with the largest RAM size available. The ASIC synthesis process was performed to compare our solution with both types of implementation. The results indicate that our RP architecture allows for a maximum matrix size of approximately 1000 rows by 1001 columns with a 24-bit word size in the chosen FPGA type. The maximum tested matrix dimension of 1000 uses more than 90 % of the available block RAM and approximately 60 % of all available slices in the FPGA while being approximately 5 times faster than a software implementation running on a CPU. In a case, when we use ASIC with a 130 nm technology, the elimination time is 4 times faster than in FPGA for a 1000 rows matrix.

Future work will focus on a new RP architecture with an external memory and a limited number of AUs.

Acknowledgment

This research was supported by the Czech Science Foundation project no. P103/12/2377.

References

[1] D. Taleshmekaeil and A. Mousavi, "The use of residue number system for improving the digital image processing," in *Signal Processing (ICSP), 2010 IEEE 10th International Conference on*, oct. 2010, pp. 775–780.

[2] A. Mirshekari and M. Mosleh, "Hardware implementation of a fast FIR filter with residue number system," in *Industrial Mechatronics and Automation (ICIMA), 2010 2nd International Conference on*, vol. 2, may 2010, pp. 312 –315.

[3] J.-C. Bajard and L. Imbert, "A full RNS implementation of RSA," *Computers, IEEE Transactions on*, vol. 53, no. 6, pp. 769 –774, june 2004.

[4] T. Güneysu and C. Paar, "Ultra high performance ECC over NIST primes on commercial FPGAs," in *Proceeding sof the 10th international workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 62–78. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85053-3-5

[5] R. T. Gregory and E. V. Krishnamurthy, *Methods and Application of Error-free Computation*. Springer Verlag, 1984.

[6] J. Buček, P. Kubalík, R. Lórencz, and T. Zahradnický, "Dedicated Hardware Implementation of a Linear Congruence Solver in FPGA," in *The 19th IEEE International Conference on Electronics, Circuits, and Systems, ICECS 2012*. Monterey: IEEE Circuits and Systems Society, 2012, pp. 689–692.

[7] R. Lórencz and M. Morháč, "A modular system for solving linear equations exactly, i. architecture and numerical algorithms," *Computers and Artificial Intelligence*, vol. 11, no. 4, pp. 351–361, 1992.

[8] M. Morháč and R. Lórencz, "A modular system for solving linear equations exactly, ii. hardware realization," *Computers and Artificial Intelligence*, vol. 11, no. 5, pp. 497–507, 1992.

[9] R. Lórencz, "New Algorithm for Classical Modular Inverse," in *Cryptographic Hardware and Embedded Systems CHES 2002*. New York: Springer, 2002, pp. 57–70.

5.5 RP5 – An ASIC Linear Congruence Solver Synthesized with Three Cell Libraries

This paper was published in *21st IEEE International Conference on Electronics, Circuits and Systems (ICECS 2014)* [A.6] The paper describes an ASIC implementation of a previously implemented FPGA linear congruence solver, part of a parallel system for solution of linear equations, and presents synthesis results for three different standard cell libraries. The previous VHDL design was adapted to three ASIC technologies (130 nm, 110 nm, and 55 nm) from two different vendors and the synthesized results were mutually compared. The maximum clock frequency and occupied area of the synthesized design were collected and analyzed for several input matrix dimensions and the maximum possible input problem size for each of the technologies was determined.

The comparison results were further used to obtain a view of design properties in higher density technologies. The results of this paper are important as input for the final considerations of the perspective of the solver of systems of linear equations, whose system design and analysis is presented in our subsequent paper **RP6**.

An ASIC Linear Congruence Solver Synthesized with Three Cell Libraries

Jiří Buček, Pavel Kubalík, Róbert Lórencz, and Tomáš Zahradnický
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9, 160 00 Prague, Czech Republic
Email: { bucekj | xkubalik | lorencz | zahradt }@fit.cvut.cz

***Abstract*— The paper describes an ASIC implementation of a previously implemented FPGA linear congruence solver, part of a parallel system for solution of linear equations, and presents synthesis results for three different standard cell libraries. The previous VHDL design was adapted to three ASIC technologies (130 nm, 110 nm, and 55 nm) from two different vendors and the synthesized results were mutually compared. The maximum clock frequency and occupied area of the synthesized design were collected and analyzed for several input matrix dimensions and the maximum possible input problem size for each of the technologies was determined. The comparison results were further used to obtain a view of design properties in higher density technologies.**

I. INTRODUCTION AND BACKGROUND

Solution of a system of linear equations (SLE) $\mathbf{Ax} = \mathbf{y}$ is a common task of linear algebra and is frequently computed in a floating point arithmetic, which involves rounding. There are numerous methods for solving linear systems of various sizes, for spare or dense systems, for differently conditioned systems, for symmetric, or positive definite systems etc., and all of them need to choose an arithmetic wherein to perform their operations. A common choice is one of the floating point (FP) arithmetics defined in the IEEE 754:2008 Standard [1] and, it is a well known fact, that using an FP arithmetic shall induce a question of numerical stability, especially in the case of large, dense, and/or ill-conditioned systems, where the input error magnification and/or roundoff errors may heavily impact or even destroy the solution of the SLE.

When rounding is undesired, it is possible to use arithmetics that do not round, such as the arithmetic of the Residual Number System (RNS) [2][3], usually at cost of an increased complexity. Parallel algorithms for solving SLEs using congruence techniques were proposed in [4][5][6] converting the input SLE into several to many independent systems of linear congruences (SLC)s, solving them independently, and reconstructing the SLE result with the Chinese Remainder Theorem, summarized in three steps:

1) Transformation of the augmented SLE matrix $\mathbf{A} \mid \mathbf{y}$ into independent linear systems $(\mathbf{A} \mid \mathbf{y}) \bmod m_i$, each with a distinct prime number modulus m_i, for $i = 1 \dots p$ being a number of moduli.
2) Solution of p independent systems of linear congruences (SLC)s in form $\mathbf{Ax} \equiv \mathbf{y} \pmod{m_i}$.
3) Reconstruction of the SLE solution vector $\mathbf{x}$ from its RNS residual representations $\mathbf{x} \pmod{m_i}$.

Using RNS increases the temporal and spatial complexity and one of the ways to go is to use a dedicated hardware Modular System (MS) performing SLE solution and exploiting natural parallelism offered by the RNS. Such MS was discussed in our previous papers [5][6][7][8].

When studying methods of solving SLEs in RNS arithmetic, a gap in the publication activity can be observed since the mid 1990's. This can be explained by the limited technology resources available at the time and thus limited sizes of instances that could be solved. At the same time, better and more sophisticated methods for solving SLEs in FP arithmetic were developed that could be performed on general purpose CPUs. The current demand for precise SLE solving is based on the fact that FP arithmetic has its limitations, and at the same time, progress in technology enables creating HW solvers that can effectively use RNS to solve SLEs for real applications. RNS is currently used in areas of digital image processing [9][10], digital signal processing [11][12][13], and in public-key [14] and elliptic curve [15][16] cryptography.

The goal of this paper is to explore the achievable area consumption and maximum speed for several ASIC technology libraries, and thus establishing the limitations on the maximum SLC instance size that can be solved. Previous work [8] indicated that the internal memory was a limiting factor and it is then interesting to compare the synthesis results in different technologies with their corresponding memory blocks.

The paper is organized as follows: Section I. introduces the reader into the problematics. Section II. provides a brief overview of the previous work. Section III. presents the architecture of the basic SLC solution unit — a Residual Processor. Section IV. summarizes the results, Section V. concludes the paper, while Section VI. describes our future efforts.

II. PREVIOUS WORK

The method of the SLE solution is based on the previous work [5][6] including methods, algorithms, and the corresponding parallel hardware architecture of the MS (Fig. 1). MS solves the SLE in RNS and therefore transforms the input SLE onto multiple independent SLCs, each with its own unique prime number modulus, solved at a distinct Residual Processor (RP). It should be noted that evaluation in each

978-1-4799-4242-8/14/$31.00 ©2014 IEEE

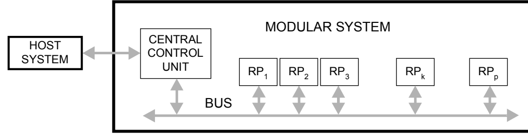


Fig. 1. Architecture of the Modular System [5]. The Central Control Unit controls solution processes and communicates with the Host System. RP_1 through RP_p are individual Residual Processors, each with its own modulus m_i for $i = 1 \ldots p$. RPs perform transformation of the input SLE into RNS, perform SLC solution mod m_i, and also back-transformation from RNS, while The Bus denotes an internal interconnection of all units within the MS.

modulus is performed independently of the others and that the addition and subtraction are carry/borrow-free across the individual moduli, and thus the computation can occur safely in parallel. Once the SLC solutions are available, they are recombined back into a solution of the SLE.

The architecture of the residual processor RP_k is depicted at Fig. 2 consisting of a Memory, Arithmetic Units AU_2, AU_3, $\ldots$, AU_{n+2} and a Control unit. RP, which was described in [7], was designed with a focus on its effective implementation in FPGA for various problem sizes with special attention to the memory architecture. The memory design was critical to the RP because of massive data access and pivoting. All important parts of the RP architecture, such as data memory, Pivot index, Pivot flags, counters, arithmetic units, inversion unit and control unit were implemented and tested in a Xilinx Virtex 6 FPGA with the largest block RAM capacity of its family, i.e. 38304 Kibits. The results showed that RP architecture with a 1000-row matrix and with a 24-bit word size in Xilinx XC6VSX475T occupied more than 90 % of the available block RAM and approximately 60 % of all available slices. This implementation was the largest possible to fit in the FPGA and ran approximately 2 times faster than a software implementation at a CPU.

The 24-bit word length was chosen as a compromise so that enough prime moduli can be generated to represent the largest number needed during solution of the system of linear equations. This follows from the Hadamard's inequality and its application on solving a linear system exactly [5].

Paper [8] implemented the same RP architecture in ASIC with a 130 nm standard cell library and compared it to the FPGA architecture described in [7]. Results in our previous paper [8] indicated that the ASIC implementation was yet 4 times faster than its FPGA counterpart.

This contribution builds on papers [7][8] and extends the ASIC implementation of our SLC solver to three different standard cell libraries for three ASIC technologies from two different vendors – Synopsys/GlobalFoundries 130 nm and Faraday/UMC 110 nm and 55 nm. The 130 nm and 110 nm libraries are high performance libraries, while the 55 nm library is a low power library. We have chosen these particular libraries mainly because of their availability including memory compilers, which are important in our design and evaluation. The last one being a low power library, it is not directly comparable in terms of speed, but it can be used to evaluate

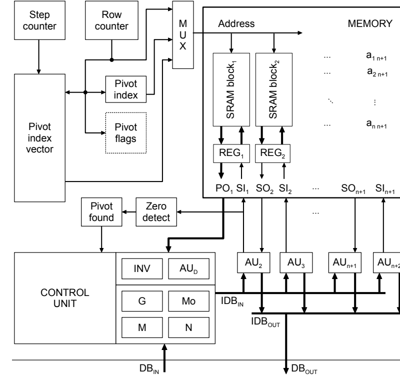


Fig. 2. Architecture of the Residual Processor. a_{ij}s represent elements of the augmented SLC matrix, AU_i stand for individual arithmetic units, PO_1 is a parallel output, $\mathrm{SI_i}$/$\mathrm{SO_i}$ denote serial inputs/outputs, $\mathrm{IDB_{IN}}$/$\mathrm{IDB_{OUT}}$ represent internal data buses, while $\mathrm{DB_{IN}}$/$\mathrm{DB_{OUT}}$ represent external data buses. INV, $\mathrm{AU_D}$, G, Mo, M, and N are auxiliary units described in [6]. Other blocks serve to implement partial pivoting.

the area savings coming from greater integration. The chosen libraries, although not the most recent ones, are sufficient to reveal technology-dependent properties of the architecture, allowing prediction of the behavior of our design in general. The output of these implementations will provide insight into the properties of the design in higher density technologies.

The next section describes the experimental results, compares them together, and analyzes the maximum clock frequency and occupied area of the synthesized RP design.

III. Architecture Overview

Each residual processor RP_k can solve systems of linear congruences (SLC)s $\mathbf{A}\mathbf{x}_k \equiv \mathbf{b} \pmod{m_k}$ and its architecture is depicted at Fig. 2 and consists of Memory, Arithmetic Units AU_2, AU_3, $\ldots$, AU_{n+2} and the Control unit.

The memory contains an augmented matrix $(\mathbf{A}|\mathbf{b}) \bmod m_k$. It consists of SRAM blocks that are created using the memory compiler specific for the ASIC library and an appropriate size block must be selected for the maximum matrix dimension n.

The arithmetic units AU are connected to the memory via Serial Inputs (SI) and Serial Outputs (SO). The leftmost elements of the matrix are read by the Control unit via the parallel bus PO_1. All AUs and the Control unit are interconnected via Internal Data Buses $\mathrm{IDB_{in}}$ and $\mathrm{IDB_{out}}$. The INV and $\mathrm{AU_D}$ units compute the modular multiplicative inverse and the determinant of $\mathbf{A} \bmod m_k$, respectively.

During elimination, partial pivoting is used, where any non-zero element can be a pivot (Zero detect block). The Pivot index block holds the row address of the pivot for the current elimination step. Pivot flags contain one bit for each row indicating whether the row contained the pivot in past elimination steps, while the Pivot index memory block contains row addresses of all pivots found up to the current elimination step. These addresses are used at the end of the elimination process for result reordering (no row swapping is done during elimination).

IV. Experimental Results

All experiments were conducted on the RP architecture shown at Fig. 2. The design was specified in VHDL, simulated in Mentor Graphics ModelSim, and synthesized using Synopsys Design Compiler.

The maximum matrix dimension n and the word length e were configurable at synthesis time using generics, while the actual matrix dimension, the modulus, and the matrix data were specified at runtime.

The solution designed for FPGA [7] was modified to obtain a solution usable for synthesis process for ASIC. The modification was focused mainly on memory interface and restrictions of the synthesis process such as a number of gate inputs. The designed ASIC architectures for various matrix dimensions were used to conduct a set of SLC solution experiments on a single RP. The SLC solution included input both data modulo reduction and matrix elimination.

In order to verify the design, we added test units to increase testability and observability of the simulated design and to verify the calculation. The test data were generated using Wolfram Mathematica and converted with a Python script into a file format suitable for the simulation. The RP was simulated with Mentor ModelSim to solve SLCs and their solutions were compared to SLCs solutions precomputed in Wolfram Mathematica. The simulation was performed for matrices up to matrix dimension $n = 100$. Matrices with dimension $n > 100$ were not simulated due to high simulation times.

After the verification of simulation correctness we started the ASIC synthesis process. Since block RAMs were not present in the ASIC library as standard cells, we generated them with a special memory generator tool coming with the library. We generated a suitable set of synchronous RAM modules with sizes to cover the expected matrix dimensions. The generated RAM modules were then instantiated according to the generic parameters from the VHDL description to implement memory matrix columns (a_{i1} to $a_{i,n+1}$).

We compared three different standard cell libraries for three ASIC technologies (130 nm, 110 nm, and 55 nm) from two different vendors. The maximum clock frequency and the occupied area of the synthesized design were collected and analyzed for several matrix dimensions. The first two technologies 130 nm and 110 nm were high speed and the last technology, the 55 nm one, was low power. Synopsys Design Compiler tools were used for all ASIC tests. In order to get a good estimate of the best achievable timing while keeping the synthesis run time reasonable, we set the synthesis effort to "medium" and defined the required minimum clock period in a compile script file. The results of our experiments are shown in Table I. The n column denotes the matrix dimension, "Area Utilization" describes an estimation of the final size, while "Frequency" describes an estimation of the maximum frequency. Both previous columns are divided into three technology size: 130 nm, 110 nm, and 55 nm (low power).

The number of clock cycles needed for the elimination process can be calculated from (1), where n stands for a matrix size and z a number of bits per word, which is also the bit length of the modulus and the data path width. For our comparisons and analyses, we chose the word size $z = 24$ and the number of words per element $q = 3$, i.e. input integer element length is 72 bits, as a reasonable compromise regarding the range of input data values. Each input integer (matrix element) is first reduced using a 24-bit modulus and all internal computation is carried out in a 24-bit modular arithmetic.

$$\mathrm{elim}(n, z) = ((z + (4z - 2))n + 3)n + 14. \qquad (1)$$

To calculate the elimination time (T_{elim}), we assume that the data is already loaded and stored in memory and the elimination process is in run. The load part takes only a small part of the SLC solution time (about 10 % for the instances considered). The elimination time on a 130 nm ASIC takes 3 ms for an $n = 100$ matrix and 380 ms for an $n = 1000$ one.

The maximum achieved frequency of the design based on matrix dimension n is shown in Fig. 3. The results show that the performance strongly depends on the type of ASIC library used. For small designs, the 55 nm library yields the best speed and the smallest area. However for $n = 300$ and larger, the 55 nm library is the slowest even though it would seem that due to its smaller pitch it should be faster. This is due to the fact that it is a low power library not optimized for performance.

Interesting effect was observed when comparing the maximum frequency between the 130 nm and 110 nm libraries for $n \geq 2000$, where the smaller technology is slower. This

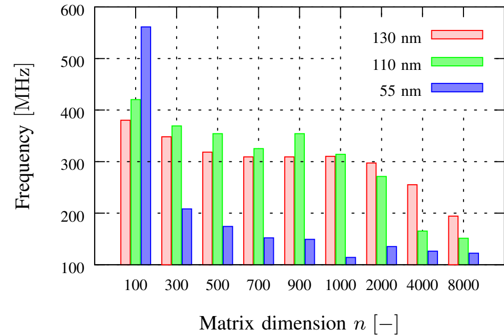


Fig. 3. Achieved frequency of the design based on matrix dimension n.

TABLE I
Synthesis results for the ASIC residual processor architecture.

n	Area Utilization (mm^2)			Frequency (MHz)		
	130 nm	110 nm	55 nm	130 nm	110 nm	55 nm
500	32	42	9	318	354	174
1000	102	117	28	310	314	114
2000	342	367	97	297	271	135
4000	1221	1260	353	255	165	151

is caused by different net delay models and different choice in cell sizes among the two libraries. Further investigation would require analysis at the layout level, however the library backends are not readily available for both libraries.

The same task was implemented on a CPU solving an SLC of dimensions 100, 500, and 1000, it takes approximately 3 ms, 424 ms, and 3.37 s, respectively calculated on Intel T9400 CPU running at 2.53 GHz with a 6144 KiB cache [17]. This shows that for $n = 1000$, our design is approximately 5 times faster.

The results show that our residual processor architecture allows for a maximum reasonable matrix size of approximately 2000 rows by 2001 columns with a word size of 24 bits in the chosen ASIC type. With larger maximum sizes, the frequency drops considerably as a result of high fan-in and fan-out of the array of AUs, and the die size grows as a result of on-chip memory required. The memory area consumption can be addressed either using on-chip dynamic RAM (which was not available in the library design kit), or better yet, by using off-chip external memory for the matrix during the elimination process. The latter option is suitable for significantly larger matrices and requires significant changes in the design.

V. Conclusion

The paper describes an implementation of a solver of systems of linear congruences in ASIC, part of a parallel system for solution of systems of linear equations. We modified the previous FPGA design for ASIC and compared three types of standard cells libraries: 130 nm, 110 nm, and 55 nm. The first two technologies were high speed, while the last one was low power. These implementations provide insight into the properties of the design in higher density technologies.

The occupied area and speed were gathered from synthesis reports. The most significant part was the memory used to store the data for calculation. The time needed to solve one SLC with 4000 congruences was 15 s and the occupied area was $3.5\,\mathrm{cm}^2$ of die. Considering a suitable die size around $1\ \mathrm{cm}^2$, the maximum matrix dimension is 1000 for the 130 and 110 nm technologies, and 2000 for the 55 nm low power technology. The 55 nm technology allowed us to use the same die space for solution of a much larger SLC, but since the technology is low power, the solution time is significantly slower than with the other two technologies.

VI. Future work

The implemented linear congruence solver, a Residual Processor (RP), will be used as a part of the Modular System (MS) for solution of sets of linear equations. Time and area complexity results of the implemented RP obtained in the paper have already been used to extrapolate properties of an MS yet to be synthesized in higher density technologies [18].

In future work we intend to focus on RP's external memory and optimization of the arithmetic unit utilization. Using external memory requires design of a memory interface, possibly limiting data throughput during the parallel read/write operations, and requires further research. The architecture with an external memory also offers opportunity to a more efficient utilization of arithmetic units during the elimination process through efficient memory interface control.

Acknowledgment

This research was supported by the Czech Science Foundation project no. P103/12/2377.

References

[1] IEEE Computer Society Standards Committee., *IEEE Standard for Floating-Point Arithmetic*, ser. ANSI/IEEE STD 754-2008. The Institute of Electrical and Electronics Engineers, Inc., 2008.

[2] A. Omondi and B. Premkumar, *Residue Number Systems: Theory and Implementation*, 1st ed. Imperial College Press, 2007, vol. 2.

[3] M. Lu, *Arithmetic and Logic in Computer Systems*. John Wiley & Sons, Inc., 2004.

[4] Ç. K. Koç, "A parallel algorithm for exact solution of linear equations via congruence technique," *Computers & Mathematics with Applications*, vol. 23, no. 12, pp. 13–24, 1992.

[5] M. Morháč and R. Lórencz, "A modular system for solving linear equations exactly, i. architecture and numerical algorithms," *Computers and Artificial Intelligence*, vol. 11, no. 4, pp. 351–361, 1992.

[6] R. Lórencz and M. Morháč, "A modular system for solving linear equations exactly, ii. hardware realization," *Computers and Artificial Intelligence*, vol. 11, no. 5, pp. 497–507, 1992.

[7] J. Buček, P. Kubalík, R. Lórencz, and T. Zahradnický, "Dedicated Hardware Implementation of a Linear Congruence Solver in FPGA," in *The 19th IEEE International Conference on Electronics, Circuits, and Systems, ICECS 2012*. Monterey: IEEE Circuits and Systems Society, 2012, pp. 689–692.

[8] J. Buček, P. Kubalík, R. Lórencz, and T. Zahradnický, "Comparison of FPGA and ASIC Implementation of a Linear Congruence Solver," in *Digital System Design (DSD), 2013 16th Euromicro Conference on*, 2013.

[9] D. Taleshmekaeil and A. Mousavi, "The use of residue number system for improving the digital image processing," in *Signal Processing (ICSP), 2010 IEEE 10th International Conference on*, oct. 2010, pp. 775–780.

[10] D. Younes and P. Steffan, "Efficient image processing application using residue number system," in *Mixed Design of Integrated Circuits and Systems (MIXDES), 2013 Proceedings of the 20th International Conference*. The Institute of Electrical and Electronics Engineers, Inc., 7 2013.

[11] G. Cardarilli, A. Nannarelli, and M. Re, "Residue number system for low-power DSP applications," in *Signals, Systems and Computers, 2007. ACSSC 2007. Conference Record of the Forty-First Asilomar Conference on*, nov. 2007, pp. 1412–1416.

[12] R. Chaves and L. Sousa, "RDSP: a RISC DSP based on residue number system," in *Digital System Design, 2003. Proceedings. Euromicro Symposium on*, sept. 2003, pp. 128 – 135.

[13] A. Mirshekari and M. Mosleh, "Hardware implementation of a fast FIR filter with residue number system," in *Industrial Mechatronics and Automation (ICIMA), 2010 2nd International Conference on*, vol. 2, may 2010, pp. 312 –315.

[14] J.-C. Bajard and L. Imbert, "A full RNS implementation of RSA," *Computers, IEEE Transactions on*, vol. 53, no. 6, pp. 769–774, 7 2004.

[15] D. Schinianakis, A. Kakarountas, and T. Stouraitis, "A new approach to elliptic curve cryptography: an RNS architecture," in *Electrotechnical Conference, 2006. MELECON 2006. IEEE Mediterranean*, may 2006, pp. 1241–1245.

[16] T. Güneysu and C. Paar, "Ultra high performance ECC over NIST primes on commercial FPGAs," in *Proceeding sof the 10th international workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 62–78. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85053-3-5

[17] L. Vondra, "System for solving linear equation systems," Dissertation thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2014. [Online]. Available: http://hdl.handle.net/10467/20222

[18] J. Buček, P. Kubalík, R. Lórencz, and T. Zahradnický, "System on chip design of a linear system solver," in *International Symposium on System-on-Chip (SoC)*, 2014 (accepted).

5.6 RP6 – Design of a Residue Number System Based Linear System Solver in Hardware

This paper was published in the *Journal of Signal Processing Systems, 87(3), 2017* [A.8].

This paper is focused on error-free solution of dense linear systems using residual arithmetic in hardware. The designed Modular System uses hardware identical Residual Processors (RP)s for solving independent systems of linear congruences and combines their solutions into the solution of the given linear system. This approach uses the residue number system which is based on the Chinese remainder theorem.

In order to efficiently exploit parallel processing and cooperation of the individual components, a hardware architecture of the Modular System with several RPs is designed. In order to verify the proposed architecture, a Xilinx FPGA with a MicroBlaze processor was used. Experimental results are obtained for an evaluation FPGA board with Virtex 6. The implemented design was used to verify correctness of the architecture and to gather time and performance data.

Results from implementation serve for subsequent theoretical analysis of the system performance for various linear system sizes and further improvement of the system. The proposed system can be useful as a special hardware peripheral or a part of an embedded system for solving large nonsingular systems of linear equations with integer, rational or floating-point coefficients with arbitrary precision.

J Sign Process Syst (2017) 87:343–356
DOI 10.1007/s11265-016-1146-1

CrossMark

Design of a Residue Number System Based Linear System Solver in Hardware

Jiří Buček[1] · Pavel Kubalík[1] · Róbert Lórencz[1] · Tomáš Zahradnický[1]

Received: 14 June 2015 / Revised: 5 May 2016 / Accepted: 20 May 2016 / Published online: 14 June 2016
© Springer Science+Business Media New York 2016

Abstract This paper is focused on error-free solution of dense linear systems using residual arithmetic in hardware. The designed Modular System uses hardware identical Residual Processors (RP)s for solving independent systems of linear congruences and combines their solutions into the solution of the given linear system. This approach uses the residue number system which is based on the Chinese remainder theorem. In order to efficiently exploit parallel processing and cooperation of the individual components, a hardware architecture of the Modular System with several RPs is designed. In order to verify the proposed architecture, a Xilinx FPGA with a MicroBlaze processor was used. Experimental results are obtained for an evaluation FPGA board with Virtex 6. Results from implementation serve for subsequent theoretical analysis of the system performance for various linear system sizes and further improvement of the system. The proposed system can be useful as a special hardware peripheral or a part of an embedded system for solving large nonsingular systems of linear equations with integer, rational or floating-point coefficients with arbitrary precision.

Keywords System of linear equations · System of linear congruences · Residue number system · Modular arithmetic · Error-free computation · FPGA

✉ Róbert Lórencz
lorencz@fit.cvut.cz

Jiří Buček
bucekj@fit.cvut.cz

[1] Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, 160 00 Prague, Czech Republic

1 Introduction

Solving a system of linear equations (SLE) is a common task in numerical mathematics. The difficulty of the solution process depends on many aspects, such as an input matrix dimension, its density, conditioning, accuracy requirements, and properties of the matrix. Solution is performed using different numerical methods and algorithms on all kinds of computational resources including PCs, GPUs, clusters, specialized hardware, and even supercomputers.

Using floating-point (FP) arithmetic, as defined in the IEEE 754:2008 Standard [1], requires the result of each operation rounded to a representable FP number, introducing a roundoff error. Input error magnification and the accumulation of rounding errors committed during solution may even destroy the result. For this reason, SLE solution shall always induce a question of numerical stability, esp. in case of a large, dense, and/or ill-conditioned system. An example field of application for error-free SLE solution is magnetohydrodynamic simulations, which involves solving large systems of differential equations (whose part is solving SLE) [2]. In some cases, solving for singular points in systems with unstable solutions, where FP arithmetic is not sufficient and traditional methods (such as the Jacobi conjugate gradient mehtod) fail.

Non-rounding arithmetics can go around rounding problems and such SLE solution processes were already implemented in special hardware. To avoid undesired rounding effects, it is possible to use a non-rounding arithmetic such as the arithmetic of the Residue Number System (RNS) [3], which, in addition to the absence of rounding, offers natural parallelism. Parallel algorithms for solving SLEs using RNS were proposed in [4–6] and are further exploited by

a dedicated hardware Modular System (MS). Such MS was discussed in [5–8].

When studying SLE solution methods in RNS arithmetic, a gap in the publication activity can be observed since the mid 1990s. This can be explained by limited technology resources available at that time and thus limited sizes of instances that could be solved. At the same time, better and more sophisticated methods for solving SLEs in FP arithmetic were developed that could be performed on general purpose CPUs. The current demand for precise SLE solution is based on the fact that FP arithmetic has its limitations, and at the same time, progress in technology enables creating hardware solvers that can effectively use RNS to solve SLEs for real applications.

RNS is currently used to accelerate computation in areas such as the digital image processing [9, 10], digital signal processing [11, 12], and cryptography [13, 14].

The paper deals with the design of a dedicated SLE solution hardware in RNS (Modular System) and provides important results for further development and verification. Architectures so far designed and testing of its implementation in FPGA and ASIC technologies were covered in papers [7, 8, 15] and [16]. These papers focused mainly on design and implementation of the Residual Processor, a solver of systems of linear congruences (SLC).

Solving SLCs is as the most important and computationally intensive part of the whole process of SLE solution, while this paper deals with Residual Processors composed into the Modular System including interconnection and integration into an SLE solving system. Such system can be useful as a special hardware peripheral attached to a host system, or a part of an embedded system requiring SLE solution. The hardware design provides a platform for analysis of a practical and functional system. The results of this analysis are important for subsequent development of the system and confirmation of the validity of the method and design process.

The paper is organized as follows: Section 1 (Introduction) introduced the reader into the context of the paper. Section 2 (Mathematical background) explains basic principles of solving SLE in RNS. Section 3 (Design of Modular System) discusses so far designed components (Residual Processors) of the designed SLE solver. Section 4 (Architecture of Modular System on FPGA) describes the system architecture of the SLE solver using the Residual Processors described earlier. Section 5 (Implementation and Experimental Results) presents the results of an implementation of the system with multiple RPs on FPGA with the ML605 prototyping board and compares the implementation's performance with a model. Section 6 (Perspectives of the Modular System) discusses possible extensions and improvement of the system, while Section 7 (Conclusion) summarizes the paper.

2 Mathematical Background

As already mentioned RNS appears to be a suitable number system for implementing certain numerical methods for error-free solving of a system of linear equations [5, 6, 17, 18]. In this section, we describe the basic mathematical principles of method solving SLE in the RNS.

Although RNS is defined on integers, we can convert systems with rational (FP) coefficients to integer coefficients. Both sides of each equation of the system can be multiplied by a suitable number that transforms all coefficients to integers. The solution of such transformed system is the same as the solution of the original SLE. In the rest of this section, we will assume only integer coefficents. Symbols used in the rest of this text and their meaning are summarized in Table 1.

Let us have a SLE of dimension n with integer coefficients

$$\mathbf{Ax} = \mathbf{b}, \tag{1}$$

where $\mathbf{A}$ is invertible and $\mathbf{b}$ is nonzero. If we denote $\mathcal{M}(\mathbf{b}) = \max(|b_i|)$ and $\mathcal{M}(\mathbf{A}) = \max(|a_{i,j}|)$, where $i, j \in \{1, 2, \ldots, n\}$, then the size of a modulus M of single-modulus residue arithmetic used in an error-free algorithm to solve (1) follows from the Hadamard's inequality and is given as [19]

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

Since the modulus M estimated using Eq. 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 suitable to use a known

Table 1 Description of used symbols.

Symbol	Meaning
$\mathbf{A}$	matrix of integer coefficients of the SLE to solve
$\mathbf{b}$	vector of integer right-hand sides
n	dimension of $\mathbf{A}$ (no. of equations in SLE)
r	number of moduli needed
p	number of Residual Processors (RPs)
m_k	k-th modulus
M	product of moduli (CRT big modulus)
$\mathbf{x}$	solution (rational) of $\mathbf{Ax} = \mathbf{b}$
$\mathbf{y}_k$	solution of $\mathbf{Ax} = \mathbf{b}$ modulo m_k
d_k	determinant of $\mathbf{A}$ mod m_k
$\mathbf{z}$	scaled solution (integers), $\mathbf{z} = \mathbf{x}d$
$\mathbf{z}_k$	scaled solution modulo m_k
$\mathbf{t}_k$	column vector $\mathbf{z}_k$ and d_k
e	word length (since Section 3)
q	number of words in each element of $\mathbf{A}$ (Section 4)

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 as long as there is a dedicated hardware architecture that allows it. The moduli must fulfill the following conditions [17, 19]:

a) $M = \prod_{i=1}^{r} m_i$ fulfills (2),

b) each m_i for $i \in \{1, 2, \ldots, r\}$ is prime.

The results of solving an SLE modulo m_i are then converted to integers according to the Chinese Remainder Theorem (CRT).

The equivalent of a SLE in modular arithmetic is a system of linear congruences (SLC) modulo m_k.

$$\mathbf{A}\mathbf{y}_k \equiv \mathbf{b} \pmod{m_k} \quad (3)$$

In order to get the solution of the SLE (1), it is necessary to solve at most r SLCs (3), $k \in \{1, 2, \ldots, r\}$, and then convert their solutions $\mathbf{y}_k$ to the solution $\mathbf{x}$ of Eq. 1.

The elements of the solution vector $\mathbf{x}$ are rational numbers, but the CRT is defined only for integers. Therfore, we need to scale the solution to obtain integers, and transform it into fractions afterwards. We make use of the fact, that the determinant $d = \det \mathbf{A}$ is an integer, and by the Cramer's rule, $\mathbf{x}\,d$ is an integer vector. Let us denote this integer vector as $\mathbf{z} = \mathbf{x}\,d$. This way, we can get $\mathbf{x}$ as fractions

$$\mathbf{x} = \frac{\mathbf{z}}{d}, \quad (4)$$

where d and the elements of $\mathbf{z}$ and are all integers.

Therefore, we multiply each $\mathbf{y}_k$ by the determinant $d_k = \det \mathbf{A} \bmod m_k$, which is computed together with solving (3). We denote this product $\mathbf{z}_k$:

$$\mathbf{z}_k = d_k\,\mathbf{y}_k \bmod m_k.$$

Because $\mathbf{z} \equiv \mathbf{z}_k \pmod{m_k}$ and $d \equiv d_k \pmod{m_k}$, we get $\mathbf{z}$ and d according to the CRT.

The method of solving (1) according to the previous conditions consists of three basic stages [5]:

Method 1

1. Input conversion. Conversion of the coefficients $a_{i,j}$ of matrix $\mathbf{A}$ and the coefficients b_i of vector $\mathbf{b}$, where $i, j \in \{1, 2, \ldots, n\}$, of SLE (1) to r systems of linear congruences (SLC) $\mathbf{A}\mathbf{y}_k \equiv \mathbf{b} \pmod{m_k}$, $k \in \{1, 2, \ldots, r\}$.
2. SLC solving. Solving r systems of SLC $\mathbf{A}\mathbf{y}_k \equiv \mathbf{b} \pmod{m_k}$ using the Gaussian elimination with pivoting [5] in modular arithmetic modulo m_k, where $k \in \{1, 2, \ldots, r\}$.

   Also the determinants $d_k = \det \mathbf{A} \bmod m_k$ are computed. Then, $\mathbf{z}_k = d_k \mathbf{y}_k \bmod m_k$ is obtained for $k = 1, 2, \ldots, r$.
3. Output conversion. Conversion of the resulting vectors $\mathbf{z}_k$ and the determinants d_k using the Mixed-Radix Conversion (MRC) algorithm [17] 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. This means that each SLC solver (from 1 to 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 SLC solvers p in a parallel system is equal to r, the SLC solver k can be allocated for computations modulo m_k: SLC solver k solves the SLC $\mathbf{A}\mathbf{y}_k \equiv \mathbf{b} \pmod{m_k}$, and computes the determinant $d_k = d \bmod m_k$, and then proceeds to compute $\mathbf{z}_k = d_k\,\mathbf{y}_k \bmod m_k$. This computation is performed simultaneously by all SLC solvers $1, 2, \ldots, r$. Since the Gaussian elimination on a matrix of dimension n requires $O(n^3)$ arithmetic steps, assuming $p = r$ we get $O(n^3 r/p) = O(n^3)$ arithmetic steps for the second step of the method.

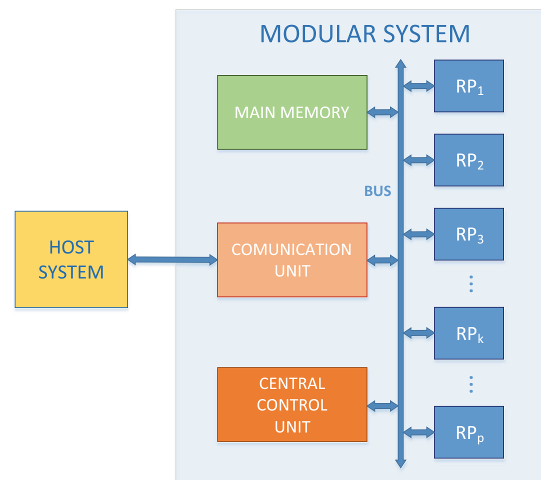


Figure 1 The architecture of the modular system [5]. The central control unit controls processes required by Method 1. RP_1 through RP_p are individual Residual Processors, each with its own modulus m_i for $i = 1 \ldots p$. RPs support all Method 1 stages i.e. perform transformation into RNS (stage 1), perform SLC solution mod m_i (stage 2), and back-transformation from RNS (stage 3). Main memory stores the input SLE and the solutions. The bus denotes an internal interconnection of all units within the modular system. The communication unit communicates with the host system.

346 J Sign Process Syst (2017) 87:343–356

Because of the easy implementation in hardware, the Gauss-Jordan (GJ) elimination algorithm was chosen for solving SLC [6]. The elimination process of one SLC according to this algorithm needs to perform so called *nonzero residue pivoting*. It is for the reason that during computation, zero value of elimination element (pivot candidate) can occur. Such a GJ elimination of SLC $\mathbf{A}\mathbf{y}_k \equiv \mathbf{b} \pmod{m_k}$ can be described by this algorithm:

Algorithm 1

Inputs: $\mathbf{A} = (a_{i,j})$ is the matrix of integers,
$\mathbf{b} = (b_i)$ is the vector of integers of (1),
m_k is the modulus of current SLC.
Outputs: $\mathbf{z}_k = \mathbf{y}_k\, d_k \bmod m_k$,
d_k is the determinant of $\mathbf{A} \bmod m_k$.
Working storage:
$\mathbf{c} = (c_1, c_2, ..., c_n)$ are the row indices of pivots,
$a^{(s)}_{i,j}$ are matrix elements in the elimination step s.
Used function:
$\operatorname{sgn}(\mathbf{c})$ is the sign of the permutation given by $\mathbf{c}$.

Input conversion.
For $i = 1, 2, \ldots, n$ **do**
For $j = 1, 2, \ldots, n$ **do**
$a^{(0)}_{i,j} = (a_{i,j}) \bmod m_k$
$a^{(0)}_{i,n+1} = (b_i) \bmod m_k$
$d_k = 1.$
SLC solving.
For $s = 1, 2, \ldots, n$ **do** (elimination steps)
Find $i \in \{1, 2, \ldots, n\}$, **such that**
$i \neq c_f$ **for all** $f < s$ **and** $a^{(s-1)}_{i,1} \neq 0.$ (pivot)
$c_s = i,$ (record pivot's index)
$d_k = d_k\, a^{(s-1)}_{i,1} \bmod m_k,$ (multiply determinant)
For $j = 1, 2, \ldots, n+1-s$ **do** (pivot's row)
$a^{(s)}_{i,j} = [a^{(s-1)}_{i,j+1}\, (a^{(s-1)}_{i,1})^{-1}] \bmod m_k;$
For $l = 1, 2, \ldots, n$ and $l \neq i$ **do**
For $j = 1, 2, \ldots, n+1-s$ **do** (other rows)
$a^{(s)}_{l,j} = [a^{(s-1)}_{l,j+1} - (a^{(s-1)}_{l,1}\, a^{(s)}_{i,j})] \bmod m_k;$

(The solution elements are in the first column, $a^{(s)}_{i,1}$.)
Let $\mathbf{y}_k = (a^{(s)}_{c_1,1}, a^{(s)}_{c_2,1}, \ldots, a^{(s)}_{c_n,1})^\mathsf{T}$, (solution reordering)
$d_k = d_k \operatorname{sgn}(\mathbf{c}) \bmod m_k,$ (sign correction)
$\mathbf{z}_k = d_k\, \mathbf{y}_k \bmod m_k$

At the end of the second step of the Method 1 we obtain the n-element vector $\mathbf{z}_k$ and the integer d_k in the SLC solver k for $k = 1, 2, \ldots, r$. We apply the MRC algorithm [17] 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 $\mathbf{t} \equiv \mathbf{t}_k \pmod{m_k}$ [20]. 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 Eq. 4.

The following Algorithm 2 represents the Output conversion step of Method 1:

Algorithm 2

Inputs:
$\mathbf{t}_k$ are the outputs from Algorithm 1
for $k \in \{1, 2, \ldots, r\}$ in the form of (5),
Outputs:
$\mathbf{x} = \frac{1}{d}\mathbf{z}$ is the solution vector of (1).
Working storage:
w is the weight of the current mixed radix digit.
Used function:

$$/a/_M = \begin{cases} a & \text{if } a < \frac{M}{2}, \\ a - M & \text{otherwise} \end{cases}$$

Output conversion.
(Mixed-Radix Conversion)
$\mathbf{t} = \mathbf{0}, \quad w = 1$
For $l = 1, 2, \ldots, r-1$ **do**
For $k = l+1, l+2, \ldots, r$ **do**
$\mathbf{t}_k = (\mathbf{t}_k - \mathbf{t}_l)\, {m_l}^{-1} \bmod m_k,$
$\mathbf{t} = \mathbf{t} + \mathbf{t}_l\, w,$
$w = w\, m_l.$
$\mathbf{t} = \mathbf{t} + \mathbf{t}_r\, w,$
(Solution finalization)
$\mathbf{t} = /\mathbf{t}/_M,$ (applied element-wise)
$\begin{bmatrix} \mathbf{z} \\ d \end{bmatrix} = \mathbf{t}, \ \mathbf{x} = \frac{1}{d}\,\mathbf{z}.$

Algorithm 2 consists of two parts. The first part is the MRC algorithm [3], the second part perform solution finalization in order to obtain the final solution of the solved SLE. The solution elements are converted to floating point representation in chosen precision. If needed, exact rational numbers can be obtained that may have large numerators and denominators in general.

Since the MRC algorithm of r moduli takes $O(r^2)$ arithmetic operations to convert one element from RNS to integer [19, 21], we have $O(nr^2)$ arithmetic operations, which are required by sequential MRC algorithm for the $(n+1)$ vector $\mathbf{t}$. This time complexity applies when we compute the final solution in floating point representation. For exact rational solution computation, the complexity will be increased due to complex operations in full precision rational numbers.

In order to illustrate the computation, an example is presented in the Appendix A.

Using Method 1 has both its pros and cons. An obvious disadvantage is the increased time and/or space complexity. An advantage is gained by exploiting the built-in parallelism, lowering the time complexity by implementing the method in hardware. Although parallel processing increases spatial complexity, computation units at the Stage 2 of Method 1 (SLC Solving) are identical and provide a time-space complexity tradeoff.

3 Design of Modular System

SLE solution in RNS performed in dedicated hardware [5] requires that the hardware is able to perform all Method 1 stages. The architecture of such system is depicted in Fig. 1.

The Modular System (MS) consists of a Central Control Unit (CCU), Comunication Unit (CU), Main Memory (MM), p hardware identical Residual Processors (RP)s – SLC solvers, and an interconnection Bus. CCU communicates through CU with an external Host System such as a computer, coordinates RP's work, and dispatches data to and from RPs. Each RP works with its own distinct prime number modulus m_i and in [5] was designed to support all 3 Method 1 stages. MM stores the input SLE and the solutions. The Bus, not necessarily implemented as a data/control bus, denotes necessary interconnection of all units within MS. The following paragraphs recapitulate so far achieved progress in papers [7, 8] and [15], all dealing with an RP architecture.

Paper [7] presents an initial RP architecture (Fig. 2) designed to perform a Gauss-Jordan (GJ) elimination upon individual SLCs mod m_i stored within the Residual Processor Memory. RPs contain specialized Arithmetic Units (AU)s interconnected with the Residual Processor Memory. This allows performing vector operations (SIMD) corresponding to the GJ elimination, which is controlled by the Control Unit. The dedicated hardware for residual pivoting also solves the zero pivot occurrence problem. Next, there are a control unit and auxilliary units (see Fig. 2), primarily for computing a multiplicative modular inverse mod m_i (INV) and the determinant (DET) of the SLC, both needed during the back-transformation performed in stage 3 of Method 1.

The Residual Processor Memory contains residues of matrix **A** and vector **b** elements. The storage of values of a row of the matrix from AU registers is performed bitwise via Serial Inputs $SI_1, SI_2, \ldots, SI_{n+1}$. Loading of values of rows from Memory to AUs is done via Serial Outputs $SO_2, SO_3, \ldots, SO_{n+1}$. The SI and SO connections with Residual Processor Memory are designed in such manner that the individual elimination steps shift the values in the rows to the left (according to GJ elimination), then the resulting vector is in the first column. The element values of the first

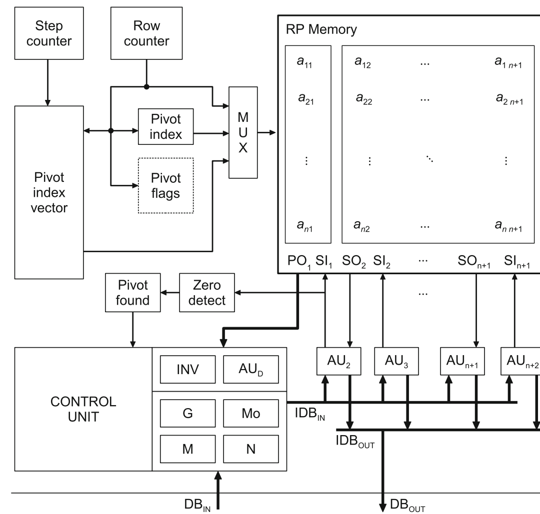


Figure 2 The Architecture of a Residual Processor. a_{ij}s represent elements of the augmented SLC matrix, AU_i stand for individual arithmetic units, PO_1 is a parallel output, SI_i/SO_i denote serial inputs/outputs, IDB_{IN}/IDB_{OUT} represent internal data buses, while DB_{IN}/DB_{OUT} represent external data buses. INV, AU_D, G, Mo, M, and N are auxilliary units described in [6]. Pivot index vector, Pivot flags, Pivot index, Zero detect, and Pivot found registers are used to support the non-zero residual pivoting [7].

matrix column are read by the Control Unit via the parallel bus PO_1. All AUs and the Control Unit are interconnected via Internal Data Buses IDB_{in} and IDB_{out}.

The above RP_k architecture can solve systems of linear congruences (SLC)s: $\mathbf{Ay}_k \equiv \mathbf{b} \pmod{m_k}$. RPs together with the Central Control Unit of the MS also fully support all conversion operations from the integer set to RNS, and by computing the determinant and multiplying the SLC solution vector $\mathbf{z}_k = d_k \, \mathbf{y}_k \bmod m_k$, they also prepare for the conversion back from RNS to the integer set. The output conversion is done in the Host System.

The Control Unit contains a finite state machine using a memory-based transition and output functions. It implements a GJ elimination algorithm as well as data input and output and allows flexibility with regard to modification and extensions.

One of the main contributions of the paper [7] is the design of new Residual Processor Memory Architecture and pivoting. The pivot column is always the first column of the matrix, and all nonzero values are equally acceptable as pivots. Search for a pivot is done sequentially; however, this search can be easily performed concurrently with write operations to the memory. The search is performed while the matrix is loaded or updated during computation. In most cases, the pivot is passed to the inversion unit (INV) long before the inverse is needed. In order for a value to be

accepted as a pivot, i) it must be nonzero, ii) the row has not contained a pivot yet, and iii) no pivot has yet been found for this elimination step.

Once the pivot is found, its row index must be stored in a pivot index vector at the address of the current elimination step. The pivot row must be flagged in order to skip it during the pivot search performed in subsequent elimination steps. If no pivot was found, the matrix is singular in this modulus.

The elimination is performed by rows. The architecture must support addressing of the pivot row first; then sequentially reduce memory matrix rows, with an exception of the pivot row which must be skipped. The first value in each row must be read in parallel. This value is either the pivot, which is inverted, or a value from a different row, which is negated.

The remaining values in each row are read bit-serial (but all values concurrently) from the MSb first. This ensures the correct order for left-shift modular multiplication and addition, and follows from the design depicted at Fig. 2.

Upon completion of the elimination process, the solution vector appears in the first column. The order of its elements corresponds to the pivot row indices and may need to be reordered. The result is therefore read out in correct order by addressing through the pivot index vector.

The following algorithm presents a simplified description of the hardware implementation of the SLC solving part (stage 2) of Algorithm 1 with focus on memory operations.

Algorithm 3

Parameters: n is matrix dimension, e is word length.

1. $s = 1$, assuming pivot found during matrix loading
2. **while** $s \leq n$ **begin**
3. Select pivot row (1 clk)
4. Read data row from memory (1 clk)
5. Multiply the pivot row with pivot^{-1} ($5e$ clk)
6. Write data and test pivot (1 clk)
7. Reset row counter (1 clk)
8. **repeat** $n-1$ **times begin**
9. Test and skip pivot row (1 clk)
10. Read data row from memory (1 clk)
11. Reduce data row using ($5e$ clk)
12. Write data and test pivot (1 clk)
13. **end repeat**
14. **end while**

The design of Arithmetic Units AU_2, AU_3, ..., AU_{n+2} and AU_D at Fig. 2 is modified in [8] from the original circuit in [6] by using strictly synchronous design. It supports computation of modulo operation on multi-word inputs, which is used when loading a new matrix into the modular system. During elimination (Algorithm 1), it computes modular multiplication and addition operations. Multiplication operations are performed using a shift-add algorithm with interleaved modulus subtraction.

The block diagram of the arithmetic unit is given in Fig. 3. The main part of the unit is the adder that performs all addition and subtraction operations including elementary additions during multiplication. The intermediate results are stored in the A register. The B register serves as a temporary storage for the values of the multiplied pivot row to be used during row reduction.

For computing the modular inverse in the INV unit, the left-shift modular inverse algorithm [22] is used.

4 Architecture of Modular System on FPGA

This section deals with design of an interconnection of the units within MS. The architecture, which is depicted at Fig. 4, is designed to allow interconnection of RPs with the Host System (HS), Main Memory, and other units to perform Stages 1 and 2 of Method 1.

In Stage 1, the augmented matrix is sent to the MS from the Host System using the Ethernet Interface. It provides a good flexibility and is supported in FPGA development systems by several performance options. The matrix in form of multi-word integer numbers is transferred and stored into the Main Memory and is therefore prepared for loading and parallel conversion to modulus m_i. Main Memory will hold the augmented matrix of the SLE to be solved and is connected to the FPGA with a (DDR3) SDRAM Controller supporting burst transfers in between the memory and other

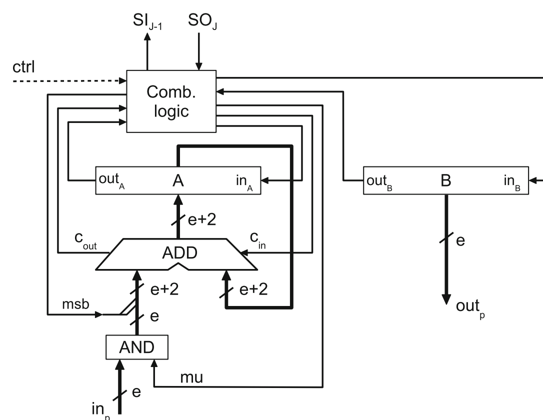


Figure 3 Arithmetic unit architecture. A and B are working registers for storing intermediate results. ADD is an $e + 2$ bit binary adder. AND is a parallel gate that controls the left input to ADD according to the value of mu and is used for shift-add multiplication and reduction. Control signals from the Control Unit (ctrl) are decoded in the Comb. logic block, which also performs data multiplexing.

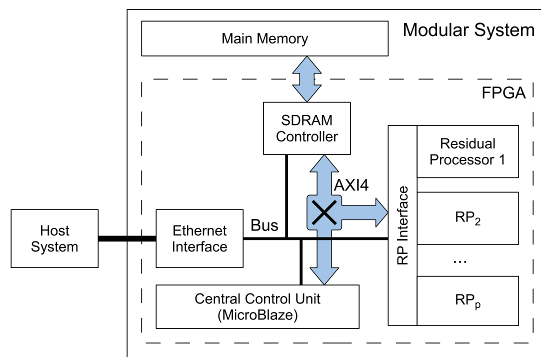


Figure 4 The modular system as a system on chip in FPGA. The central control unit is a MicroBlaze soft-processor. RP_1 through RP_p are individual Residual Processors, each with its own modulus m_1 to m_p. RPs are connected using an internal bus and form a common AXI4 bus master peripheral. The AXI4 crossbar switch is used for a high speed interconnect, while the Bus is used for a lower speed programmed I/O.

parts of the system. A high-throughput channel is created using a high performance interconnect (AXI4 in crossbar configuration) between the SDRAM Controller and a set of RPs. RPs share a common data channel from Main Memory since they always get the same data. During loading of a matrix, each RP applies the modulo operation on the data with its modulus m_i. The set of RPs is connected using a common bus-master AXI4 peripheral (the RP Interface). The system contains a Xilinx MicroBlaze processor as the Central Control Unit (CCU) for overall communication, data loading, and synchronization control. The MicroBlaze processor was chosen as a convenient way to create a prototype implementation including network protocol handling. Although MicroBlaze is not a final solution, it is sufficient for performing experiments and evaluation of basic properties of the proposed system.

The CCU uses a network protocol for communication and data transfer from the Ethernet Inteface to the Main Memory. After receiving the matrix data, it starts the process of loading this data from Main Memory to the RPs. This is done by initiating a bus master read by the RP Interface from a specified address in memory. The RP Interface broadcasts the data to all RPs for parallel loading and conversion. After loading and converting all the data (Stage 1 of Method 1 (Input Conversion)), the RPs automatically start solving their SLCs independently (Stage 2 of Method 1 (SLC Solving)). The CCU can either wait for the RPs to complete or perform other tasks such as communication to load another instance from the HS.

The SLE solving process according to Method 1 can be implemented on the architecture in Fig. 4 using two algorithms synchronized by data communication over the Ethernet Interface (Eth). The algorithms, Algorithm 4 for the Host System, and Algorithm 5 for the Modular System, including their cooperation are shown in Fig. 5.

The system is now ready to perform experiments and evaluate the experimental results.

5 Implementation and Experimental Results

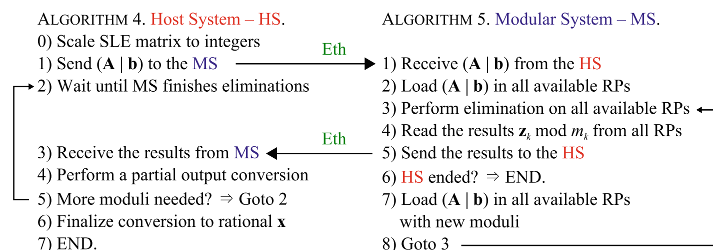
The designed architecture of the MS (see Fig. 4) was implemented in FPGA and verified by simulation and hardware prototyping. The implementation and testing platform used for the development was a Xilinx ML605 board with a Virtex-6 LX240T FPGA, 1 GiB DDR3 SDRAM memory, gigabit Ethernet interface, and other peripherals. The MS is controlled using a MicroBlaze soft-processor inside the FPGA.

The MicroBlaze processor controls the transfer of data between Main Memory and the RPs. External DDR3 SODIMM was the Main Memory used to store the augmented matrix of the SLE, and the processor program code.

The hardware (RPs, the RP bus-master and slave interface) was described in VHDL, while the software running on MicroBlaze was written in C. The MicroBlaze processor is an IP core included in the development tool. Xilinx ISE and Embedded Development Kit (EDK) development tools were used to describe, synthesize, and implement the SoC architecture into FPGA.

The selected FPGA platform allowed for a system with the number of parallel RP units $p = 1, 2, \ldots, 5$, and matrix dimensions $n = 20, 100, 200$. Each matrix element

Figure 5 Algorithms for solving a SLE synchronized by data communication over the Ethernet Interface (Eth). Note that partial output conversion (HS step 4) is interleaved with the computation in the MS.



is $q = 3$ words long, where each word is $e = 24$ bits long. Each modulus m_i is also 24-bit all internal computation is done in a 24-bit modular arithmetic, and the following text assumes these values. Parameters p, n, q, e are configurable at synthesis time. The maximum attainable matrix dimension n depends on the number of RPs p implemented on one FPGA. For up to $n = 200$, we could attain the default clock frequency of the MicroBlaze processor of 100 MHz. Without the MicroBlaze processor impairing the design timing, the attainable frequency for $n = 100$ was 166 MHz, and for $n = 200$, 154 MHz.

The RP peripheral attachment to the AXI4 crossbar was created using a bus master template from Xilinx EDK. The RPs are connected to the rest of the system via a FIFO. The networking code for the MicroBlaze processor was based on a TCP echo server example contained in the EDK and was extended to enable data transfer in between the Main Memory and an RP peripheral. Time measurement was done using a dedicated timer peripheral and reported using a serial interface.

5.1 Experimental Results

The testing data for our experiments were generated using Wolfram Mathematica. We have generated several SLE instances together with their solutions for verification. Data were transfered between PC and the tested system over Ethernet using the ncat[1] tool.

The times needed for processing of individual steps, mainly for load, elimination, and result read are collected. The resulting measurements are presented in Table 2. The column Load describes time needed for loading data from the Main Memory to all RP units including reduction modulo m_i, Elim describes time needed for computing the solution vector in p moduli in parallel (for example, modulo $m_1, \ldots, m_p$) by GJ elimination in all p RPs, and Read denotes time needed to retreive the results from the corresponding number of RPs into the Main Memory. Loading and elimination are performed with all RPs in parallel, while the read operation is performed sequentially. Table 2 shows that most of the time is spent in elimination. The chosen platform (Xilinx Virtex-6 LX240T) allowed us to implement a maximum of 2 RPs for $n = 200$.

Table 3 presents the area occupation measured in lookup tables (LUTs) and block RAM of the Virtex-6 device used. In order to simplify the table, the RAM blocks are presented as equivalent number of 18 Kibit blocks denoted as RAMB18E1, any RAMB36E1 is counted twice.

The gathered performance data from a real implementation are important for evaluation of the whole modular system operation as it reflects the real behavior including external parts (e.g. the DDR3 memory). The measured time and area complexity are used for subsequent analysis, which is the topic of the next section.

[1] Ncat 5.51 (http://nmap.org/ncat)

Table 2 Load, elimination, and read times for the MS architecture with multiple RPs.

n	Load [ms]	Elim [ms]	Read [ms] for a number of RPs				
			1	2	3	4	5
20	0.13	0.51	0.015	0.025	0.036	0.047	0.058
100	2.18	12.60	0.054	0.104	0.154	0.204	0.254
200	7.46	50.41	0.103	0.202	–	–	–

5.2 Analysis of the Experimental Results

The number of clock cycles needed for load and elimination process can be calculated by the Eqs. 6 and 7 which were presented in [8]. These equations express the behavior of a model of the RP without accounting for its surroundings, i.e. interface to other parts of the MS.

$$\mathrm{load}(n, e, q) = ((2n + 2e + 5)q + 7)n + 1, \tag{6}$$

$$\mathrm{elim}(n, e) = ((5e - 2)n + 3)n + 14. \tag{7}$$

The $\mathrm{load}(n, e, q)$ equation describes the number of cycles needed to accept the matrix of n by $n + 1$ elements, each having q words e bits long. It includes the cycles needed to reduce the matrix modulo an e-bit modulus, and storing the matrix in the RP Memory. The $\mathrm{elim}(n, e)$ equation describes the number of cycles an RP needs for the GJ elimination to compute the solution vector, assuming the data already loaded and stored in the RP Memory.

The experimental measured times (Table 2) were compared the with the theoretical estimation from Eqs. 6 and 7. For this comparison, the clock period was assumed to be 10 ns (corresponding with the implementation). The elimination process is largely independent of the control by the

Table 3 FPGA (Xilinx xc6vlx240t) area occupation of MS architecture with multiple RPs.

n	p	1	2	3	4	5	1 (RP only)
20	LUTs	11932	15527	18838	22124	25021	3595
	BRAM	43	66	89	112	135	23
100	LUTs	20525	32897	45222	57347	66499	12372
	BRAM	123	226	329	432	535	103
200	LUTs	30630	52886	–	–	–	18698
	BRAM	223	426	–	–	–	203

The last column is only the RP without the rest of the MS. BRAM is the number of block RAM primitives computed as the number of RAMB18E1 plus 2 × RAMB36E1

MicroBlaze Central Control Unit and the time measurement was burdened only with a small error caused by reading state of an external timer.

The measured load times however were $\approx$ 2.7 times longer than the estimation. For example, for $n = 200$, the estimated load time was 2.73 ms according to Eq. 6, but 7.46 ms was measured. (See also Table 4 described in the next section.) Load times were first estimated with an assumption that all data were available when needed. In the real system, control and synchronization overhead was added to the loading times, causing this load time growth. Even though data were read from Main Memory using AXI4 master read transfers, it was necessary to read them in blocks (of 32 words) under MicroBlaze software control. When loading data, the RP(s) compute remainders of loaded data modulo the assigned modulus. Thus some form of throttling or flow control had to be implemented and in our case, this was done in software. The associated overhead could be decreased by improving the master read data throttling without the need for software control. Even with the current overhead, the load portion of the time was small compared to the elimination.

6 Perspectives of the Modular System

In this section, a performance analysis of a technically realistic hypothetical Modular System configuration is presented. The experimental data gathered in the previous sections will be used to evaluate performance of the overall projected system.

The number and magnitude of all moduli can be derived from M in Eq. 2. From this bound, a number of e-bit prime moduli needed to express the solution can be derived. The number of moduli r is generally higher than the number of RPs p in MS, and thus it is necessary to process r by p moduli in parallel.

For the subsequent analysis, a SLE size of $n = 20$ to 2000, and the number of RPs in the system $p = 25$ to 1000 is assumed. The operating frequency of the MS is $f_{cl} = 100$ MHz, this figure is taken from the implementation, and is determined by the MicroBlaze processor used. With a different clock configuration, the frequency could be higher, therefore 100 MHz is a conservative estimation. The data throughput between the HS and MS is 1 Gibit/s.

Table 4 contains the times of loading, elimination, and result read of a single RP. Numbers printed in cursive signify esimated values according to Eqs. 6 and 7, numbers printed in bold signify measured values, while the remaining numbers (for $n = 1000$ and 2000) were extrapolated from the actually measured times.

In order to analyze the performance of the system, it was necessary to fractionate the time in a communication and computation part. First, the communication part is analyzed.

In Table 5 we present the amount of data that is sent from the Host System to the Modular System to transfer the SLE augmented matrix, and also the amount of data received as the solution, i.e. MS to HS. The received data depends on the number of moduli actually used that is always less than the maximum number of moduli r.

From Table 5 we can see that for the considered cases, and thus the required data amounts, the estimated times are in the order of hundreds of milliseconds. If needed, the times could be improved by using a faster data connection (e.g. a 10 Gibit/s Ethernet or PCI Express).

Some FPGAs, such as the Xilinx Zynq platform, contain hard-core integrated processor (ARM Cortex A9). This can be beneficial as the hardware processor has greater performance than the Microblaze soft-processor. Central Control Unit of the Modular System runs on this processor and therefore can benefit of more computing power for handling communication protocols.

Next, we analyze the time taken by the computation part. We use the values for a single RP from Table 4, and extrapolate based on the number of RPs p from 25 to 1000 and n from 100 to 2000. The results of the extrapolation are presented in Table 6.

The r column in Table 6 denotes the maximum number of moduli needed for the solution. The loading and elimination process is done in parallel by p RPs. If more than p moduli is needed, RPs are assigned a new moduli set and the process is repeated (c.f. Algorithms 4 and 5). The results show that solving larger systems ($n \geq 1000$) in reasonable

Table 4 *Estimated*, **Measured**, and Extrapolated Times for 1 RP.

n	Load [ms]		Elim [ms]		Read [ms]	Total [ms]
	Est	M/Ex	*Est*	M/Ex		
20	*0.06*	**0.13**	*0.47*	**0.51**	**0.015**	0.66
100	*0.77*	**2.18**	*11.8*	**12.60**	**0.054**	14.84
200	*2.73*	**7.46**	*47.2*	**50.41**	**0.103**	57.96
1000		~ 174		~ 1260	0.5	1434
2000		~ 688		~ 5039	1	5728

Table 5 Ethernet transfer data size and estimated time (1 Gibit/s).

		Max Size [KiB]			Data Transfer
n	r	HS to MS	MS to HS	Total	Time [ms]
100	314	89	93	182	1
200	632	353	372	725	6
1000	3208	8798	9408	18206	149
2000	6457	35174	37853	73027	598

Table 6 Solution time for different sizes and numbers of RP units.

		Solution time for a MS with p RP [s]				
n	r	$p = 50$	$p = 100$	$p = 200$	$p = 500$	$p = 1000$
100	314	0.110	0.06	0.04	0.04	0.04
200	632	0.80	0.43	0.25	0.14	0.14
1000	3208	93.55	47.57	24.58	10.78	6.19
2000	6457	746.0	376.2	191.3	80.3	43.3

time (units and tens of seconds) can be achieved by using parallel systems with $p = 100$ and more RPs. Although the times may seem high for the considered SLE dimensions, it must be noted that the solution occurs without rounding errors. By comparing the computation and communication parts of the time, it is evident that even for $p = 1000$ RPs, and $n = 2000$, the communication time is in orders of magnitude less than computation.

The conversion process consists of two parts, namely transformation into mixed-radix digit represenatation, and rational number computation. In order to estimate the processing power needed, experiments were performed using a PC (a single core of Intel Core i7–2640M runnning at 2.8 GHz, code compiled using GCC (4.8.5). The results are presented in Table 7. From the table and comparison with Table 6 it can be seen that one CPU core is able to process approx. $p = 100$ RP units. The conversion process can be easily parallelized either by vector elements, or by moduli.

The performance can be further improved by fine-tuning the design of the architecture. The architecture can be implemented in ASIC [8], [15]. Using more advanced technology nodes and construction (e.g. 3D stacking for memory connection) could speed up the system up to 10 times. By employing a larger word size, the performance can be further improved.

The number of moduli needed is always less than the theoretical upper bound considered in our data. The amount of moduli needed is detected during MRC conversion. In most cases, around a half of the maximum number of moduli is needed. Considering the mentioned possible improvements,

Table 7 MRC part of the output conversion time for different SLE sizes, **measured for 1 CPU core**, extrapolated.

		MRC conversion time [s]			
n	r	1 core	4 cores	16 cores	64 cores
100	314	**0.046**	0.012	0.003	0.0007
200	632	**0.406**	0.102	0.025	0.0063
1000	3208	**51.05**	12.76	3.19	0.798
2000	6457	**413.5**	103.38	25.84	6.46

Table 8 Performance comparison between different SLE solver implementations using RNS for $n = 1000$ and $p = 50$.

	Solution time [s]		
	of single SLC	of whole SLE	
Implementation	$p = 1$	$p = 1$	$p = 50$
Plain C	5	16040	320
Mathematica	15	48120	960
Our system	1.47	4700	94

Compared to sequential solution time ($p = 1$), MRC conversion not included

the SLE $n = 2000$ could be solved on a MS with $p = 500$ to 1000 RPs in a time under 1 second. Such a result would be useful for a number of applications requiring solving SLEs without rounding errors.

The output conversion from the RNS into the rational number set, which is performed by the HS, also contributes (not necessarily significantly) to the overall time and it is therefore important to perform the conversion efficiently. One such parallel approach is described in [23]. In our case ($n = 2000$), we would need a computing cluster with approx. 100 cores to keep the conversion time approx. 1 second.

We can also implement the RNS-based algorithms for error-free solving of SLEs in plain C or Mathematica. Table 8 presents a comparison of these implementations. Plain C and Mathematica versions were run on a single core of Intel Core i7–2640M runnning at 2.8 GHz and extrapolated with the assumption of $p = 50$ and $r = 3208$ to obtain the solution time. Even though our system is a proof-of-concept that runs at substantially lower frequency (100 MHz), it is still approx. 3.4 times faster.

7 Conclusion

Solving dense systems of linear equations (SLE) without loss of precision is an important problem of numerical mathematics. Error-free solution of SLEs is important in several fields of physical modeling and analysis (for example, [2]), where traditional methods struggle to provide valid solution. The Modular System (MS) architecture of an SLE solver using residue arithmetic to avoid rounding errors is presented. MS was described in VHDL, designed, and a prototype was implemented on a Xilinx ML605 development board with a Virtex 6 FPGA with a 1 GiB DDR3 memory and a gigabit Ethernet interface, which was used to transfer data between the Host System and the evaluation board. RNS was chosen as a way to enable parallel processing that

overcomes problems encountered in computing with very large precision operands.

The implemented design was used to verify correctness of the architecture and to gather time and performance data with a limited number of a maximum of 5 parallel Residual Processors (RP) and 200 equation SLEs. Measured time also contained a control overhead of starting data transfers and start/end of the elimination process. Elimination times agreed with the estimation and high load time overhead was identified and future improvements suggested. In the current state our system can solve a single system of linear congruences (SLC) with $n = 200$ equations in approx. 58 ms. Time and area complexity of the real implementation including external memory is useful for future development of a complete system with many residual processors. Based on the prototype implementation, we may extrapolate that for $n = 1000$, with 50 processors, our architecture would solve a SLE without rounding errors in approx. 94 s, compared to software with 320 s. This is a promising result with potential for future improvements and optimizations including ASIC implementation that would enable order-of-magnitude faster computation, enabling the solution of SLEs with larger dimensions. Such a system can be useful as a hardware peripheral attached to a Host System, or part of an embedded system needing error-free SLE solution. The important feature of this system is the ability to scale performance just by attaching additional modules in the form of hardware-identical RPs.

Acknowledgment This research was supported by the Czech Science Foundation project no. P103/12/2377.

Appendix A: Examples

Let us have a system of 3 linear equations given by the augmented matrix:

$$\begin{pmatrix} 3 & -1 & 2 & 1 \\ 1.5 & 3 & -2 & -1 \\ 0.5 & -1 & 1.5 & 2 \end{pmatrix}$$

First, we convert the system to an equivalent system with integer coefficients. In this case, it suffices to multiply rows 2 and 3 by 2. The solution to this system is the same as before. This matrix is the input to Method 1, and thereby also to Algorithm 1:

$$(\mathbf{A}|\mathbf{b}) = \begin{pmatrix} 3 & -1 & 2 & 1 \\ 3 & 6 & -4 & -2 \\ 1 & -2 & 3 & 4 \end{pmatrix}$$

Let us now choose the set of moduli $(m_1, m_2, m_3) = (5, 7, 11)$ for the RNS representation. For example, the element $a_{2,2} = 6$ has the RNS representation $(1, 6, 6)$ modulo $(5, 7, 11)$. This representation is unique modulo $M = \prod_{i=1}^{r} m_i = 5 \times 7 \times 11 = 385$.

Then, we compute the residues of $(\mathbf{A}|\mathbf{b})$ mod m_1, m_2, m_3, which gives the augmented matrices of three systems of linear congruences (SLCs). This operation denoted in Algorithm 1 as well as in Method 1 as Input conversion.

$$(\mathbf{A}|\mathbf{b}) \bmod 5 = \begin{pmatrix} 3 & 4 & 2 & 1 \\ 3 & 1 & 1 & 3 \\ 1 & 3 & 3 & 4 \end{pmatrix}$$

$$(\mathbf{A}|\mathbf{b}) \bmod 7 = \begin{pmatrix} 3 & 6 & 2 & 1 \\ 3 & 6 & 3 & 5 \\ 1 & 5 & 3 & 4 \end{pmatrix}$$

$$(\mathbf{A}|\mathbf{b}) \bmod 11 = \begin{pmatrix} 3 & 10 & 2 & 1 \\ 3 & 6 & 7 & 9 \\ 1 & 9 & 3 & 4 \end{pmatrix}$$

Next, we will solve the individual SLCs in their respective moduli (SLC Solving). We will show the process for $m_2 = 7$.

In the first elimination step, a pivot is found $a_{1,1}^{(0)} = 3$. Its row index is stored in $c_1 = 1$, and the determinant intermediate value is $d_2 = 3$.

The pivot's inverse[2] is $3^{-1} \bmod 7 = 5$ and the pivot's row is multiplied with this inverse: $a_{1,j}^{(1)} = [a_{1,j+1}^{(0)} \times 3] \bmod 7$. The first row is thereby shifted one element to the left (no information is lost, the discarded value would be a constant 1).

$$a_{i,j}^{(0)} = \begin{pmatrix} \mathbf{3} & 6 & 2 & 1 \\ 3 & 6 & 3 & 5 \\ 1 & 5 & 3 & 4 \end{pmatrix} \rightarrow \begin{pmatrix} 2 & 3 & 5 & \cdot \\ 3 & 6 & 3 & 5 \\ 1 & 5 & 3 & 4 \end{pmatrix}$$

Further in this elimination step, all other rows ($l = 2, 3$) are reduced using the adjusted pivot's row: $a_{l,j}^{(1)} = [a_{l,j+1}^{(0)} - (a_{l,1}^{(0)} a_{1,j}^{(1)})] \bmod 7$. Again, the elements are by this process shifted to the left, discarding unnecessary zeros.

$$a_{i,j}^{(1)} = \begin{pmatrix} 2 & 3 & 5 & \cdot \\ 0 & 1 & 4 & \cdot \\ \mathbf{3} & 0 & 6 & \cdot \end{pmatrix}$$

In the second elimination step, a pivot is found $a_{3,1}^{(1)} = 3$. Its row index is stored in $c_2 = 3$, and the determinant intermediate value is $d_2 = 3 \times 3 \bmod 7 = 2$. The pivot's inverse is $3^{-1} \bmod 7 = 5$ and the pivot's row is multiplied with this inverse: $a_{3,j}^{(2)} = [a_{3,j+1}^{(1)} \times 5] \bmod 7$.

$$\begin{pmatrix} 2 & 3 & 5 & \cdot \\ 0 & 1 & 4 & \cdot \\ 0 & 2 & \cdot & \cdot \end{pmatrix}$$

[2]The modular multiplicative inverse can be computed by one of the variants of the Extended Euclidean Algorithm; for further reference, see [21] or [17], for example.

Then, all other rows ($l = 1, 2$) are reduced using the adjusted pivot's row: $a^{(2)}_{l,j} = [a^{(1)}_{l,j+1} - (a^{(1)}_{l,1}\, a^{(2)}_{3,j})] \bmod 7$. Again, the elements are by this process shifted to the left, discarding unnecessary zeros.

$$a^{(2)}_{i,j} = \begin{pmatrix} 3 & 1 & \cdot & \cdot \\ \mathbf{1} & 4 & \cdot & \cdot \\ 0 & 2 & \cdot & \cdot \end{pmatrix}$$

In the third and final elimination step, a pivot is found $a^{(2)}_{2,1} = 1$. Its row index is stored in $c_3 = 2$, and the determinant intermediate value is $d_2 = 2 \times 1 \bmod 7 = 2$. The pivot's inverse is $1^{-1} \bmod 7 = 1$ and the pivot's row is multiplied with this inverse: $a^{(3)}_{2,j} = [a^{(2)}_{2,j+1} \times 1] \bmod 7$.

$$\begin{pmatrix} 3 & 1 & \cdot & \cdot \\ 4 & \cdot & \cdot & \cdot \\ 0 & 2 & \cdot & \cdot \end{pmatrix}$$

The other rows ($l = 1, 3$) are reduced as in the previous steps: $a^{(3)}_{l,j} = [a^{(2)}_{l,j+1} - (a^{(2)}_{l,1}\, a^{(3)}_{2,j})] \bmod 7$

$$a^{(3)}_{i,j} = \begin{pmatrix} 3 & \cdot & \cdot & \cdot \\ 4 & \cdot & \cdot & \cdot \\ 2 & \cdot & \cdot & \cdot \end{pmatrix}$$

The solution $\mathbf{y}_2$ is now in the first column, but not in the correct order, since the pivots were found in the order of $\mathbf{c} = (1, 3, 2)$. Therefore, the solution is reordered to get $\mathbf{y}_2 = (3, 2, 4)^\mathsf{T}$.

For the same reason, the sign of the computed determinant must be corrected. In the index vector $\mathbf{c}$, an odd number of pair swaps is needed to create the ordered sequence $(1, 2, 3)$. Therefore, $\mathrm{sgn}(\mathbf{c}) = -1$ and the determinant's sign is corrected $d_2 = 2 \times (-1) \bmod 7 = 5$.

The solution is then multiplied by the determinant to get $\mathbf{z}_2 = \mathbf{y}_2\, d_2 \bmod 7 = (3, 2, 4)^\mathsf{T} \times 5 \bmod 7 = (1, 3, 6)^\mathsf{T}$.

Similarly, the solutions and determinants in the other moduli are computed, giving $\mathbf{z}_1 = (0, 0, 2)^\mathsf{T}$, $d_1 = 4$ and $\mathbf{z}_3 = (2, 1, 7)^\mathsf{T}$, $d_3 = 8$.

Next, we will perform the Output conversion, the third stage of Method 1. First, the intermediate results are written in the form of $\mathbf{t}_k = \begin{bmatrix} \mathbf{z}_k \\ d_k \end{bmatrix}$

$$[\mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3] = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 3 & 1 \\ 2 & 6 & 7 \\ 4 & 5 & 8 \end{bmatrix}$$

The values $\mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3$ are now converted according to Algorithm 2. First, we will show the conversion of the determinant d, whose RNS digits are the last elements of $\mathbf{t}_k$, i.e. $d = \det \mathbf{A}$ has the RNS representation $(d_1, d_2, d_3) = (4, 5, 8)$ modulo $(5, 7, 11)$.

l	Operation	m_1=5	m_2=7	m_3=11
		[4]	5	8
1	-4		-4	-4
1	=		1	4
1	$*m_1^{-1}$		$*3$	$*9$
	=		[3]	3
2	-3			-3
2	=			0
2	$*m_2^{-1}$			$*8$
	=			[0]

During the computation of Algorithm 2, mixed-radix digits (4, 3, 0) are computed, shown in boxes in the table above. (In fact, the first digit 4 is just taken from $\mathbf{t}_1$). The mixed-radix digit weights are $(1, m_1, m_1 m_2) = (1, 5, 5 \times 7)$. The value of the determinant is thus $d = 4\times1+3\times5+0\times5\times7 = 19$. Because $d < \frac{M}{2}$, i.e. $19 < \frac{385}{2}$, it is positive.

A negative number is converted the same way. For example, the first element of $\mathbf{z}$ has the RNS represenatation $(0, 1, 2)$:

l	Operation	m_1=5	m_2=7	m_3=11
		[0]	1	2
1	-0		-0	-0
1	=		1	2
1	$*m_1^{-1}$		$*3$	$*9$
	=		[3]	7
2	-3			-3
2	=			4
2	$*m_2^{-1}$			$*8$
	=			[10]

The mixed-radix representation is thus (0, 3, 10), and the value is $0+3\times5+10\times5\times7 = 365$, which is more than $\frac{M}{2}$, therefore it is negative. The correct value can be computed by subtracting M, yielding $365 - 385 = -20$.

The same process is applied on all members of $[\mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3]$, getting $(\mathbf{z}, d)^\mathsf{T} = (-20, 45, 62, 19)^\mathsf{T}$.

The final step is to get the solution vector $\mathbf{x} = \frac{\mathbf{z}}{d} = (-\frac{20}{19}, \frac{45}{19}, \frac{62}{19})^\mathsf{T}$. We can now verify that $\mathbf{x}$ is indeed the solution of the SLE $(\mathbf{A}|\mathbf{b})$:

$$\begin{pmatrix} 3 & -1 & 2 \\ 3 & 6 & -4 \\ 1 & -2 & 3 \end{pmatrix} \cdot \begin{pmatrix} -\frac{20}{19} \\ \frac{45}{19} \\ \frac{62}{19} \end{pmatrix} = \begin{pmatrix} 1 \\ -2 \\ 4 \end{pmatrix}$$

References

1. IEEE Computer Society Standards Committee (2008). IEEE Standard for Floating-Point Arithmetic. ANSI/IEEE STD 754-2008. IEEE.

2. Skála, J., & Bárta, M. (2012).
3. Garner, H.L. (1959). The residue number system. In *Papers presented at the the March 3–5, 1959, Western Joint Computer Conference. IRE-AIEE-ACM '59 (Western), New York, NY, USA, ACM* (pp. 146–153).
4. Koç, Ç.K. (1992). A parallel algorithm for exact solution of linear equations via congruence technique. *Computers & Mathematics with Applications*, *23*(12), 13–24.
5. Morháč, M., & Lórencz, R. (1992). A modular system for solving linear equations exactly, i. Architecture and numerical algorithms. *Computers and Artificial Intelligence*, *11*(4), 351–361.
6. Lórencz, R., & Morháč, M. (1992). A modular system for solving linear equations exactly, ii. Hardware realization. *Computers and Artificial Intelligence*, *11*(5), 497–507.
7. Buček, J., Kubalík, P., Lórencz, R., & Zahradnický, T. (2012). Dedicated Hardware Implementation of a Linear Congruence Solver in FPGA. In *The 19th IEEE international conference on electronics, circuits, and systems, ICECS 2012, Monterey, IEEE* (pp. 689–692).
8. Buček, J., Kubalík, P., Lórencz, R., & Zahradnický, T. (2013). Comparison of FPGA and ASIC implementation of a linear congruence solver. In *16th Euromicro Conference on Digital System Design (DSD), 2013* (pp. 284–287).
9. Taleshmekaeil, D., & Mousavi, A. (2010). The use of residue number system for improving the digital image processing. In *10th International Conference on Signal Processing (ICSP), 2010 IEEE* (pp. 775–780).
10. Younes, D., & Steffan, P. (2013). Efficient image processing application using residue number system. In *Proceedings of the 20th International Conference on Mixed design of integrated circuits and systems (MIXDES), 2013, IEEE.*
11. Cardarilli, G., Nannarelli, A., & Re, M. (2007). Residue number system for low-power DSP applications. In *Signals, Systems and Computers, 2007. ACSSC 2007. Conference Record of the Forty-First Asilomar Conference on* (pp. 1412–1416).
12. Mirshekari, A., & Mosleh, M. (2010). Hardware implementation of a fast FIR filter with residue number system. In *2nd International Conference on Industrial Mechatronics and Automation (ICIMA), 2010*, (Vol. 2 pp. 312–315).
13. Schinianakis, D., Kakarountas, A., & Stouraitis, T. (2006). A new approach to elliptic curve cryptography: an RNS architecture. In *Electrotech. Conf., 2006. MELECON 2006. IEEE Mediterranean* (pp. 1241–1245).
14. Güneysu, T., & Paar, C. (2008). Ultra high performance ECC over NIST primes on commercial FPGAs. In *Proceedings of the 10th international workshop on cryptographic hardware and embedded systems. CHES '08* (pp. 62–78). Berlin, Heidelberg: Springer.
15. Buček, J., Kubalík, P., Lórencz, R., & Zahradnický, T. (2014). An ASIC linear congruence solver synthesized with three cell libraries. In *The 21th IEEE international conference on electronics, circuits, and systems, ICECS* (p. 2014). France: Marseille.
16. Buček, J., Kubalík, P., Lórencz, R., & Zahradnický, T. (2014). System on chip design of a linear system solver. In *2014 international symposium on system-on-chip proceedings* (pp. 1–6). Piscataway: IEEE.
17. Gregory, R.T., & Krishnamurthy, E.V. (1984). Methods and application of error-free computation, Springer.
18. Young, D.M., & Gregory, R.T. (1973). *A Survey of Numerical Mathematics. Addison-Wesley Series in Mathematics edn* Vol. 2: Addison-Wesley Publishing Company, Inc.
19. Newman, M. (1967). Solving exquations exactly. *National Bureau of Standarts*, *71B*, 171–179.
20. Koç, Ç.K., & Güvenç, A. (1994). B.b.g.: exact solution of linear equations on distributed-memory multiprocessors. *Parallel Algorithms and Applications*, *3*, 135–143.
21. Knuth, D.E. (1998). *The Art of Computer Programming, Seminumerical Algorithms. Mass. Third edition edn* Vol. 2: Addison-Wesley Publishing Company, Inc.
22. Lórencz, R. (2002). New algorithm for classical modular inverse. In *Cryptographic hardware and embedded systems, CHES*, (Vol. 2002 pp. 57–70). New York: Springer.
23. Vondra, L. (2014). System for solving linear equation systems. PhD thesis, Czech Technical University in Prague.

Jiří Buček graduated from the Faculty of Electrical Engineering of the Czech Technical University in Prague in 2002. Currently he is a Ph.D. student at the Faculty of Information Technology, Czech Technical University in Prague. His research interests are modular arithmetic, cryptography, and FPGA.

Pavel Kubalík graduated from the Faculty of Electrical Engineering of the Czech Technical University in Prague in 2002. He obtained his Ph.D. degree in 2007 at the Faculty of Information Technology, Czech Technical University in Prague. His research interests are fault-tolerant design, high-speed wireless networks, signal processing algorithms for GNSS, modular arithmetic, and FPGA.

Róbert Lórencz graduated from the Faculty of Electrical Engineering of the Czech Technical University in Prague in 1981. He obtained his Ph.D. degree in 1990 at the Institute of Measurement and Measuring Methods, Slovak Academy of Sciences in Bratislava. Currently he is a full professor of Information Security at the Faculty of Information Technology, Czech Technical University in Prague. His research interests are alternative arithmetic for numerical computation, arithmetic units, cryptology in general, and the design of PUF and TRNG.

Tomáš Zahradnický graduated at the Faculty of Electrical Engineering of the Czech Technical University in Prague in 2003. He received his Ph.D. in 2011 at the Department of Computer Science at the same faculty. Tomas works as a Chief Security Officer at Boxtrap s.r.o. and his interests include computer security, reverse engineering, and penetration testing.

Chapter 6

Conclusions

6.1 Summary

Error-free (exact) solution of systems of linear equations is a demanding taks that requires the usage of special arithmetic operations. The Residue Number System is used as a means to represent arbitrary precision numbers, it comes however with its specific issues that must be addressed. Solving systems of linear equations in modular arithmetic using Gauss-Jordan elimination requires implementation of individual operations as well as a hardware architecture that can implement the algorithm.

Papers **RP1** and **RP2** deal with individual operations of multiplication and inverse in modular arithmetic. The papers were written with mainly cryptographic applications in mind, however the type of operation is equivalent and of high importance for other applications of modular computation as well.

In **RP1**, we introduce a new modified carry-save encoding scheme for Montgomery multiplication that reduces the number of saved carry bits. It enables us to trade speed with area, aiming to optimize the architecture for FPGAs with dedicated adder carry chain logic. We have tested the architecture for several operand lengths k and word sizes w, where $w = 1$ corresponds to the normal carry-save encoding. The area occupation decreases with increasing w, as expected. The clock frequency decreases however, as the prolonged carry chains enlarge the critical path delay. The time-area product has its minimum in $w = 4$ for $k = 1024$ and 512, and in $w = 8$ for $k = 2048$. For the minimum time-area product, the area saving more than 20 % relatively to the conventional carry-save encoding ($w = 1$).

In **RP2**, we have implemented the Subtraction-free Almost Montgomery Inverse algorithm in FPGA and compared it to two different architectures for the traditional Almost Montgomery Inverse algorithm with subtractions. We have synthesized the generic designs using bit lengths of $n = 64, 128, 162$ and 256 bits, and gathered data about the speed and area from the implementation tools.

Implementation results show that the Subtraction-free AMI algorithm [37] is suitable for hardware implementation and its implementation is equally fast as the implementation of AMI with two subtractors, yet about 13–17% smaller in area. The Subtraction-free AMI

implementation is equally small as the implementation of AMI with one subtractor, yet it is about 25% faster. We have observed that for smaller bit lengths, the Subtraction-free AMI implementation is both smaller and faster.

Our papers **RP3** to **RP5** deal with design and technology aspects of the hardware solver of systems of linear congruences. Finally, **RP6** analyzes the solver of systems of linear equations as a complete system including hardware peripherals (Residual Processors) and the Host System managing conversions and communication tasks to facilitate solving SLEs.

The SLC solvers were implemented and verified in Xilinx Virtex-6 FPGA exploiting the internal memory and the parallel processing of the FPGA. The architecture was also adapted and synthesized for various ASIC technologies to examine the properties of the parallel architecture in ASIC and gather knowledge about its limits.

The system for solving systems of linear equations was implemented on the Xilinx ML605 prototyping board with a Virtex-6 LX240T FPGA. A system-on-chip architecture was created with several RPs as peripherals and a MicroBlaze soft-processor as a central controller. The system was prototyped with $p = 1$ to 5 RPs and maximum linear system size $n = 20$ to 200, up to the maximum capacity of the FPGA type used. Tables 6.1 and 6.1 summarize the time and area implementation results, respectively.

n	Load [ms]	Elim [ms]	Read [ms] for a number of RPs					Total (1 RP) [ms]
			1	2	3	4	5	
20	0.13	0.51	0.015	0.025	0.036	0.047	0.058	0.66
100	2.18	12.60	0.054	0.104	0.154	0.204	0.254	14.84
200	7.46	50.41	0.103	0.202	-	-	-	57.96

Table 6.1: Load, elimination, and read times for the MS architecture with multiple RPs. The Total architecture is the sum of each row for the case of 1 RP.

n	p	1	2	3	4	5	1 (RP only)
20	LUTs	11932	15527	18838	22124	25021	3595
	BRAM	43	66	89	112	135	23
100	LUTs	20525	32897	45222	57347	66499	12372
	BRAM	123	226	329	432	535	103
200	LUTs	30630	52886	-	-	-	18698
	BRAM	223	426	-	-	-	203

Table 6.2: FPGA (Xilinx xc6vlx240t) area occupation of MS architecture with multiple RPs. The last column is only the RP without the rest of the MS. BRAM is the number of block RAM primitives computed as the number of RAMB18E1 plus 2 × RAMB36E1.

6.2 Contributions of the Dissertation Thesis

The main contribution of this thesis is the exploration of the architecture of the solver of systems of linear equations in modular arithmetic with respect to the underlying platform properties (FPGA, ASIC technologies). The system-dependent optimization has shown the strengths and weaknesses of using residue arithmetic and RNS in particular for solving systems of linear equations. The requirements for error-free solution is a major factor in the overall complexity as it influences the operand lengths and thus number of moduli and solution time.

For the process of solving systems of linear congruences, a new synchronous internal memory architecture was proposed with support for pivoting and reordering of solution elements.

FPGA prototype system using an embedded processor and peripheral was designed enabling verification of the architecture and function testing. System integration (Modular System) of one central processor with several peripherals (Residual Processors) enables running multiple solvers using multiple moduli for computation of the SLC solution.

Communication complexity was analyzed and total solution times estimated for several posisble problem dimensions. The suitability and performance of the Gauss-Jordan elimination method using RNS depends on several factors. Time and area complexity was studied depending on the maximum linear system size, input number lengths and the prime modulus length. The input number length, i.e. the dynamic range, and the maximum number of equations, directly influence the maximum possible number of bits in needed to represent the exact solution, and therefore also the number and size of individual moduli in RNS.

A system designer must decide on the word length of the internal processing units, and this sets the limits on the maximum value of the modulus, and consequently, the maximum number of moduli available (in the case of many moduli, only prime numbers are suitable). For example, 16 bit word size may be not enough, as there are only 3030 prime numbers of this length, and this may be limiting the maximum system size and dynamic range that can be solved. We have determined that 24 bit moduli give enough choice, and with a conservative assumption of 3 words per element, we need aproximately as many moduli as 3 times the number of equations.

The resulting system architecture permits error-free solution of dense systems of linear equations of sizes of approx. 2000 equations in reasonable configuration using contemporary technology.

6.3 Future Work

The author of the dissertation thesis suggests to explore the following:

- Improve the Residual Processor's internal computation by improving the critical path delay and possibly also different organization from the current serial-parallel multiplication and bit-serial, operand-parallel memory access.

- Design a memory access system for sparse matrices, as the current memory system supports only dense matrices effectively.
- Design hardware support for back conversion (MRC), currently handled in the host system (in software).
- Exploit advanced ASIC technology including novel interconnection methods with memory. Three-dimensional chip design may be highly benefitial.
- Exploit different architecture possibilities for the arithmetic unit utilization – compute segment-wise, for example.

Bibliography

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

[2] Morháč, M.; Lórencz, R. A Modular System for Solving Linear Equations Exactly, II. Hardware Realization. *Computers and Artificial Intelligence*, volume 11, no. 5, 1992: pp. 497–507.

[3] Skála, J.; Bárta, M. LSFEM implementation of MHD numerical solver. *arXiv preprint arXiv:1206.2730*, 2012.

[4] Montgomery, P. Modular Multiplication Without Trial Division. *Mathematics of Computation*, volume 44, no. 170, April 1985: pp. 519–521.

[5] Menezes, A. J. *Elliptic curve public key cryptosystems*, volume 234. Kluwer Academic Pub, 1993.

[6] Eğecioğlu, O.; K. Koç, C. K. Exponentiation Using Canonical recoding. *Theoretical Computer Science*, volume 129, no. 2, 2004: pp. 407–417.

[7] Newman, M. Solving equations exactly. *Journal of Research of the National Bureau of Standards*, volume 71, 1967: pp. 171–179.

[8] Karatsuba, A.; Ofman, Y. Multiplication of Many-Digital Numbers by Automatic Computers. *Doklady Akad. Nauk SSSR*, volume 145, 1962: pp. 293–294, translation in Physics-Doklady 7, 595-596, 1963.

[9] Cook, S. On the minimum computation time of functions. *Doctoral diss., Harvard U., Cambridge, Mass*, volume 1, 1966.

[10] Schönhage, A.; Strassen, V. Schnelle multiplikation grosser zahlen. *Computing*, volume 7, no. 3-4, 1971: pp. 281–292.

[11] Walter, C. Montgomery Exponentiation Needs No Final Subtraction. *Electronics Letters*, volume 35, no. 21, October 1999: pp. 1831–1832.

[12] Hachez, G.; Quisquater, J.-J. Montgomery Exponentiation with no Final Subtractions: Improved Results. In *Cryptographic Hardware and Embedded Systems – CHES 2000*, *Lecture Notes in Computer Science*, volume 1965, Springer-Verlag, 2000, pp. 293–301.

[13] Knuth, D. E. *Seminumerical Algorithms*, *The Art of Computer Programming*, volume 2. Addison Wesley, 1969.

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

[15] Savaş, E.; Koç, C. K. The Montgomery Modular Inverse – Revisited. *IEEE Transaction on Computers*, volume 49, no. 7, 2000: pp. 763–766.

[16] Lórencz, R. New algorithm for classical modular inverse. In *Cryptographic Hardware and Embedded Systems-CHES 2002*, Springer, 2003, pp. 57–70.

[17] Zhang, W.; Betz, V.; Rose, J. Portable and scalable FPGA-based acceleration of a direct linear system solver. *ACM Trans. Reconfigurable Technol. Syst.*, volume 5, no. 1, Mar. 2012: pp. 6:1–6:26, ISSN 1936-7406, doi:10.1145/2133352.2133358. Available from: `http://doi.acm.org/10.1145/2133352.2133358`

[18] He, C.; Qin, G.; Lu, M.; et al. Group-alignment based accurate floating-point summation on FPGAs. In *ERSA'06: Proc. of the 6th International Conference on Engineering of Reconfigurable Systems and Algorithms*, Citeseer, 2006, pp. 136–142.

[19] He, C.; Qin, G.; Ewing, R. E.; et al. High-precision blas on fpga-enhanced computers. *Proceedings of ERSA 2007*, 2007: pp. 107–116.

[20] Dou, Y.; Lei, Y.; Wu, G.; et al. FPGA accelerating double/quad-double high precision floating-point applications for ExaScale computing. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, New York, NY, USA: ACM, 2010, ISBN 978-1-4503-0018-6, pp. 325–336, doi:10.1145/1810085.1810129. Available from: `http://doi.acm.org/10.1145/1810085.1810129`

[21] Dussé, S. R.; Kaliski Jr, B. S. A Cryptographic Library for the Motorola DSP56000. In *Advances in Cryptology - EUROCRYPT '90*, edited by I. Damgård, Springer-Verlag, 1991, pp. 230–244.

[22] Koç, C. K.; Acar, T.; Kaliski Jr, B. S. Analyzing and comparing Montgomery multiplication algorithms. *Micro, IEEE*, volume 16, no. 3, 1996: pp. 26–33, ISSN 0272-1732, doi:10.1109/40.502403.

[23] Koç, C. K. RSA Hardware Implementation. Technical report, RSA Data Laboratories, 1995.

[24] Blum, T.; Paar, C. High-radix Montgomery modular exponentiation on reconfigurable hardware. *Computers, IEEE Transactions on*, volume 50, no. 7, 2001: pp. 759–764, ISSN 0018-9340, doi:10.1109/12.936241.

[25] Orup, H. Simplifying Quotient Determination in High-Radix Modular Multiplication. In *Proc. 12th IEEE Symposium on Computer Arithmetic*, edited by S. Knowles; W. H. McAllister, IEEE Computer Society, 1995, pp. 193–199.

[26] Takagi, N. A Radix-4 Modular Multiplication Hardware Algorithm for Modular Exponentiation. *IEEE Transactions on Computers*, volume C-41, no. 8, August 1992: pp. 949–956.

[27] Kornerup, P. High-Radix Modular Multiplication for Cryptosystems. In *Proc. 11th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society, 1993, pp. 277–283.

[28] Daly, A.; Marnane, W. Efficient architectures for implementing montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, ACM, 2002, pp. 40–49.

[29] Savaş, E.; Tenca, A. F.; Koç, C. K. A scalable and unified multiplier architecture for finite fields GF(p) and GF(2^m). In *Cryptographic Hardware and Embedded Systems-CHES 2000*, Springer, 2000, pp. 277–292.

[30] Tenca, A. F.; Koç, C. K. A scalable architecture for modular multiplication based on Montgomery's algorithm. *Computers, IEEE Transactions on*, volume 52, no. 9, 2003: pp. 1215–1221, ISSN 0018-9340, doi:10.1109/TC.2003.1228516.

[31] Savas, E.; Tenca, A. F.; Ciftcibasi, M. E.; et al. Multiplier architectures for GF(p) and GF(2^n). In *Computers and Digital Techniques, IEE Proceedings-*, volume 151, IET, 2004, pp. 147–160.

[32] Mclvor, C.; McLoone, M.; McCanny, J. V. Fast Montgomery modular multiplication and RSA cryptographic processor architectures. In *Signals, Systems and Computers, 2003. Conference Record of the Thirty-Seventh Asilomar Conference on*, volume 1, IEEE, 2003, pp. 379–384.

[33] Gutub, A. A.-A.; Tenca, A. F.; Koç, C. K. Scalable VLSI architecture for GF(p) Montgomery modular inverse computation. In *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, 2002, pp. 46–51, doi:10.1109/ISVLSI.2002.1016874.

[34] Savaş, E.; Naseer, M.; Gutub, A.-A.; et al. Efficient unified Montgomery inversion with multibit shifting. *IEE Proceedings-Computers and Digital Techniques*, volume 152, no. 4, 2005: pp. 489–498.

[35] Savas, E. A Carry-Free Architecture for Montgomery Inversion. *IEEE Transactions on Computers*, volume 54, no. 12, Dec. 2005: pp. 1508–1519.

[36] Hlaváč, J.; Lórencz, R. Ordinary Modular Inverse Using AMI–Hardware Implementation. In *Proc. IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems–DDECS*, volume 3, 2003, pp. 309–310.

[37] Lórencz, R.; Hlaváč, J. Subtraction-free Almost Montgomery Inverse Algorithm. *Information Processing Letters*, volume 94, no. 1, 2005: pp. 11–14.

[38] Hlaváč, J.; Lórencz, R. Arithmetic Unit for Computations in GF(p) with the Left-Shifting Multiplicative Inverse Algorithm. In *Architecture of Computing Systems – ARCS 2013*, *Lecture Notes in Computer Science*, volume 7767, Springer Berlin Heidelberg, 2013, ISBN 978-3-642-36423-5, pp. 268–279, doi:10.1007/978-3-642-36424-2_23.

[39] Shoup, V. NTL: A library for doing number theory. 2007, `www.shoup.net/ntl`.

Reviewed Publications of the Author Relevant to the Thesis

[A.1] Buček, J.; Lórencz, R. Montgomery Multiplication on FPGA with Modified Carry-Save Encoding. *International Conference on Signals and Electronic Systems, ICSES'04*, Poznan, PTETiS, 2004, pp. 313–315, ISBN 83-906074-7-6.

[A.2] Buček, J. Montgomery Multiplication Implementations in GF(p). *Počítačové Architektúry a Diagnostika (PAD 2004)*, CVUT FEL Praha, 2004, pp. 117–121.

[A.3] Buček, J.; Lórencz, R. Comparing subtraction-free and traditional AMI. *Design and Diagnostics of Electronic Circuits and systems, DDECS'06* IEEE, pp. 97–99, 2006.

The paper has been cited in:

- Guajardo, J.; Güneysu, T.; Kumar, S.; Paar, C.; Pelzl, J. Efficient Hardware Implementation of Finite Fields with Applications to Cryptography, *Acta Applicandae Mathematica*, 93(1), September 2006, Kluwer Academic Publishers, pp. 75-118. *(citation indexed in WoS SCI, Scopus)*
- Schinianakis, D.M.; Fournaris, A.P.; Michail, H.E.; Kakarountas, A.P.; Stouraitis T. An RNS implementation of an F_p elliptic curve point multiplier, *IEEE Transactions on Circuits and Systems I: Regular Papers*, 56(6), June 2009, pp. 1202-1213. *(citation indexed in WoS SCI, Scopus)*
- Mohammadi, M.; Molahosseini, A.S. Efficient design of Elliptic Curve Point Multiplication based on fast Montgomery modular multiplication. *Computer and Knowledge Engineering (ICCKE), The 3rd International Conference on*, 2013, IEEE, pp. 424-429. *(citation indexed in WoS SCI, Scopus)*
- Mohan, A.P.V. RNS in Cryptography. *Residue Number Systems: Theory and Applications*, 2016, Birkhäuser, Springer International Publishing, pp. 263-347. *(citation indexed in Scopus)*

[A.4] Buček, J.; Kubalík, P.; Lórencz, R.; Zahradnický, T. Dedicated hardware implementation of a linear congruence solver in FPGA. *19th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 689–692, 2012.

[A.5] Buček, J.; Kubalík, P.; Lórencz, R.; Zahradnický, T. Comparison of FPGA and ASIC Implementation of a Linear Congruence Solver. *16th Euromicro Conference on Digital System Design (DSD)* 2013.

[A.6] Buček, J.; Kubalík, P.; Lórencz, R.; Zahradnický, T. An ASIC linear congruence solver synthesized with three cell libraries. In: *Electronics, Circuits and Systems (ICECS), 21st IEEE International Conference on*, pp. 706–709, 2014.

[A.7] Buček, J. and Kubalík, P. and Lórencz, R., and Zahradnický, T., System on Chip Design of a Linear System Solver, *2014 International Symposium on System-on-Chip Proceedings*, Piscataway, US, 2014, ISBN 9781479968909,

[A.8] Buček, J.; Kubalík, P.; Lórencz, R.; Zahradnický, T. Design of a Residue Number System Based Linear System Solver in Hardware. In: *Journal of Signal Processing Systems*, 87(3), pp.343-356. 2017.

Remaining Publications of the Author Relevant to the Thesis

[A.9] Buček, J.; Kubalík, P.; Lórencz, R.,; Zahradnický, T., System Design of an FPGA Linear Solver, *Proceedings of the Work in Progress Session held in connection with the 40th EUROMICRO Conference on Software Engineering and Advanced Applications and the 17th EUROMICRO Conference on Digital System Design*, Linz, AT, 2014, ISBN 978-3-902457-40-0.

[A.10] Buček, J.; Lórencz, R., Subtraction Free Almost Montgomery Inverse in ASIC and FPGA, *Proceedings of Workshop 2006*, Praha, CZ, 2006, pp. 260–261, ISBN 80-01-03439-9.

[A.11] Buček, J., Montgomery Multiplication Implementations in GF(p), *Počítačové architektúry a diagnostika*, Bratislava, SK, 2004, pp. 117–121, ISBN 80-969202-0-0.

[A.12] Buček, J.; Lórencz, R., GF(p) Montgomery Multiplication for Cryptosystems on FPGA, *Proceedings of Workshop 2005*, Praha, CZ, 2005, pp. 282–283, ISBN 80-01-03201-9.

[A.13] Buček, J., Montgomery Multiplication on FPGA, *POSTER 2004*, Praha, CZ, 2004, pp. IC5.

Remaining Publications of the Author

[A.14] Jeřábek, S.; Buček, J.; Schmidt, J.,; Novotný, M., Emulator of Contactless Smart Cards in FPGA, *Proceedings of the 6th Mediterranean Conference on Embedded Computing (MECO 2017)*, Bar, 2017, pp. 96–99, ISBN 978-1-5090-6741-1.

[A.15] Buček, J.; Novotný, M.,; Štěpánek, F., Practical Session: Differential Power Analysis for Beginners, *Hardware Security and Trust*, 2017, pp. 77–91, ISBN 978-3-319-44316-4.

[A.16] Buchovecká, S.; Lórencz, R.; Kodýtek, F.,; Buček, J., True Random Number Generator Based on ROPUF Circuit, *Proceedings of 19th Euromicro Conference on Digital System Design DSD 2016*, Los Alamitos, CA, US, 2016, pp. 519–523, ISBN 978-1-5090-2816-0.

[A.17] Kodýtek, F.; Lórencz, R.; Buček, J.,; Buchovecká, S., Temperature Dependence of ROPUF on FPGA, *Proceedings of 19th Euromicro Conference on Digital System Design DSD 2016*, Los Alamitos, CA, US, 2016, pp. 698–702, ISBN 978-1-5090-2816-0.

[A.18] Kodýtek, F.; Lórencz, R.,; Buček, J., Improved ring oscillator PUF on FPGA and its properties, pp. 55–63, *Microprocessors and Microsystems*, 47, November, 2016, ISSN 0141-9331.

[A.19] Bartík, M.; Buček, J., A Low-Cost Multi-Purpose Experimental FPGA Board for Cryptography Applications, *2016 IEEE 4th Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, Piscataway, NJ, US, 2016, ISBN 978-1-5090-4473-3.

[A.20] Bartík, M.; Buček, J., A Low-Cost Unified Experimental FPGA Board for Cryptography Applications, *TRUDEVICE 2016 Final Conference*, Barcelona, 2016, pp. 75–80.

[A.21] Štěpánek, F.; Buček, J.,; Novotný, M., Differential Power Analysis under Constrained Budget: Low Cost Education of Hackers, *Proceedings of 16th Euromicro Conference on Digital System Design*, Piscataway, US, 2013, pp. 645–648, ISBN 978-0-7695-5074-9.

[A.22] Buček, J.; Fornůsek, T.; Moňok, M.; Altman, T.,; Lórencz, R., Analýza zabezpečení komunikace bezkontaktních čipových karet, (Research Report), 2011, pp. 40.

[A.23] Lórencz, R.; Zahradnický, T.,; Buček, J., Forenzní analyzátor pro operativní analýzu, *Sborník příspěvků XXXII. konference EurOpen*, Plzeň, CZ, 2008, pp. 51–58, ISBN 978-80-86583-14-3.

[A.24] Buček, J.; Hlaváč, J.; Lórencz, R.,; Matušková, M., Cost-Effective Architectures for RC5 Brute Force Cracking, pp. 61–66, *Acta Polytechnica*, 45 (2), August, 2005, ISSN 1210-2709.

[A.25] Matušková, M.; Hlaváč, J.; Buček, J.,; Lórencz, R., RC5 Brute Force Cracking Engine, *Proceedings of the Sixth International Scientific Conference Electronic Computers and Informatics ECI 2004*, Košice, SK, 2004, pp. 259–264, ISBN 80-8073-150-0,

[A.26] Buček, J.; Lórencz, R., Speedup of Computation using Accelerators, *Workshop 2004*, Praha, CZ, 2004, pp. 370–371, ISBN 80-01-02945-X,

[A.27] Buček, J., Metodika pro připojování numerických akcelerátorů k nadřazeným systémům, *Počítačové Architektury & Diagnostika PAD 2003*, Brno, CZ, 2003, pp. 36–37, ISBN 80-214-2471-0,

[A.28] Kubalík, P.; Buček, J., FPGA Implementation of USB 1.1 Device Core, *Proceedings of Workshop 2003 (online)*, Praha, CZ, 2003, pp. 304–305, ISBN 80-01-02708-2,