Are Coarse Grain Reconfigurable Architectures Suitable for Cryptography?

Daniel Mesquita, Lionel Torres, Michel Robert, Gilles Sassatelli, Fernando Gehm Moraes

Are coarse grain reconfigurable architectures suitable for cryptography?

Daniel Mesquita, Lionel Torres, Michel Robert, Gilles Sassatelli
LIRMM – Université Montpellier II – France
{mesquita, torres, robert, sassate}@lirmm.fr

Fernando Moraes
PUCRS – Porto Alegre - Brazil
moraes@inf.pucrs.br

Abstract

Cryptographic algorithms are increasingly used in personal transaction through Internet network, smart card and telecommunication applications. Those algorithms are well known for their high complexity data computing. This paper focuses on Montgomery algorithm, which computes modular multiplication efficiently. This operation is the most time consuming part of public-key cryptosystems. A comparison among different implementations of the Montgomery algorithm in different architectures (GPP[1], DSP, fine grain FPGA, coarse grain reconfigurable architecture [CGRA]) is presented. It is shown that coarse grain reconfigurable architecture might be good candidate to implement and to execute Montgomery multiplication. The paper also details the CGRA chosen to implement the referred algorithm.

1. Introduction

Encryption and decryption operations of most public-key cryptosystems consists, including RSA 0 and Elliptic Curves Cryptography (ECC) [2], in modular arithmetic. For instance, the RSA and ECC performance is essentially determined by the efficiency of modular multiplication implementation, which is the basis of modular exponentiation.

Being a time consuming process, modular arithmetic requires specific computing methods, and a hardware implementation is essential to guaranty high performances.

In terms of methods to design modular multiplication, there are many different algorithms, as the ones proposed by Barret [3], Booth [4], Blakley [5] and Montgomery [6]. In this work, the Montgomery algorithm was chosen, due its efficiency and flexibility to be adapted to alternative architectures.

Nevertheless, in addition to good algorithms, it is mandatory an efficient hardware implementation to achieve good performance. As result of the increasing integrated circuit's transistor density, measured by gate count, it is possible to implement complete systems in a single chip (System-on-a-Chip – SoC), merging processors, memory and reconfigurable logic. In this work it is shown that a cryptosystem can be implemented in a SoC where the reconfigurable part can be either a fine or a CGRA.

Coarse grain reconfigurable architectures uses, in general, reconfigurable data path units (rDPUs) with large path width (i.e.32bits), in contrast with the bit-level approach of FPGAs. This allows mapping applications with a higher level of abstraction, using compilation techniques instead logical synthesis. Besides, the CGRAs, for applications like telecommunications and multimedia, are more efficient in terms of area and performance comparing with GPPs. Also, related with FPGAs, CGRAs are more energy-efficient and reduce considerably the reconfigurability overhead [22].

This paper is organized as follows. Section 2 introduces the Montgomery algorithm, used in some public-key algorithms. Section 3 presents briefly the state-of-art in hardware implementations of the Montgomery method. Section 4 shows the Systolic Ring [7], a CGRA that can be used to runs cryptosystems. Finally, Section 5 presents the results obtained and discusses some further works.

2. Montgomery Algorithm

The Montgomery algorithm computes:

$$Mont(A,B,M) = A \times B \times r^{-1} \bmod M \qquad (1)$$

without making use of any division. The algorithm constraints are:

- Both *A* and *B* must be smaller than *M;*
- *r* must be prime relatively to *M.*
- and *A, B* and *M* are represented as:

$$A = \sum_{i=0}^{k-1} a_i \times r^i; \quad B = \sum_{i=0}^{k-1} b_i \times r^i; \quad M = \sum_{i=0}^{k-1} m_i \times r^i$$

Targeting digital systems design, it is logical to take $r = 2^t$ (*t* is given by the word size in bits of an architecture, in this case, *t=1*). Even though the algorithm works for any value of *r*. But taking *r* as a 2 power's makes the computation fast and the restriction $gcd^2(M,r) = 1$ (second constraint above) is respected by simply choosing M odd.

[1] General Purpose Processor

[2] Greatest Common Divisor

In order to compute equation (1), it is considered the fact that a residual would not change even if a multiple number of modulo is added. See in Figure 1 related to Figure 2 (a bit-level version of the Montgomery algorithm). During multiplication of A and B, M is added so that the least significant bit side could be zero. Lets suppose that R is the intermediate result of (1), as the modulo M is odd, M is added if R is also odd (r_0=1), otherwise zero is added (i.e. $r_0 \times M$, represented Figure 1 as "M or 0").Usually, the binary-base version of the algorithm (Figure 2) is more suitable to be implemented in fine grained devices, such as FPGAs.

To obtain the correct result, i.e., to eliminate the r^{-1} factor and take only the $A \times B$ mod M value, it is necessary to post multiply the first result by 2^{2n} mod M, or to pre-multiply A and B by 2^n mod M.

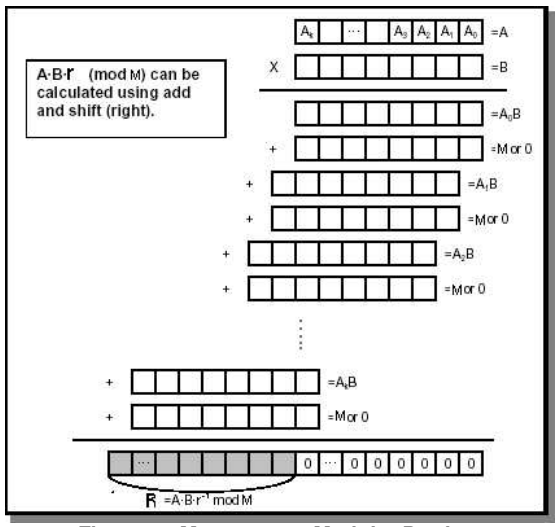

Figure 1 - Montgomery Modular Product

```
Mont (A, B, M)
  R ← 0;
  for i = 0 to n − 1
      R ← R + a_i × B;
      R ← (R + r_0 × M) / 2;
  Return R;
end.
```

Figure 2 – Bit-level Montgomery algorithm

The Montgomery algorithm works for any base, which makes possible to obtain good performances also in processors, DSP, and coarse grain reconfigurable architectures. The Figure 3 shows a word-level implementation of Montgomery algorithm. In this representation A, B and M are represented as follows:

$$A = \sum_{i=0}^{k-1} a_i \times \beta^i;\ B = \sum_{i=0}^{k-1} b_i \times \beta^i;\ M = \sum_{i=0}^{k-1} m_i \times \beta^i$$

Where k, t and β are:

$k = number\ of\ words$

$t = size\ in\ bits\ of\ each\ word$

$\beta = 2^t$

In this form of Montogmery algorithm, the modular inverse of the M's last significant digit (Figure 3, the $(-m_0)^{-1}$ factor) is pre-calculated (by the GPP, in the SoC context) with the extended Euclidean algorithm [23].

```
Mont (A, B, M)
  int R ← 0;
  for i = 0 to k − 1
      q_i ← (r_0 + a_i × b_0) × (−m_0)^-1 mod β;
      R ← R + a_i × B + q_i × M;
      R ← R / β;
  Return R;
end.
```

Figure 3 - Word-level Montgomery algorithm

Comparing the algorithms above to classical modular multiplication [8], the Montgomery algorithm reverses the order of treating the digits of the multiplicand A, performs a shift down instead up on each iteration, and does an addition rather than a subtraction. These differences allow several simplifications in the combinational logic. For other versions of this algorithm, the reference [9] gives a good comparison among different implementations of Montgomery algorithm.

3. State-of-Art

There are many implementations of the Montgomery algorithm in the literature, but in most cases the results shown are hidden behind the execution of the cryptosystem altogether. For instance, the paper [10] describes modular exponentiation using the Montgomery algorithm in a FPGA, but does not explicit the performance of only one multiplication of two large numbers. The reference [11] also gives results of modular exponentiation in FPGA devices.

Reference [12] presents the performance of Montgomery modular multiplication implemented in a FPGA, applied to ECC. Meanwhile, [13] and [14] brings ASICs designs for Montgomery product, but once more without showing clearly the results for a single product between two large integers.

On the other hand, references [15] (see Table 1, the FPGA line) and [16] are publications concerning FPGA implementations of modular multiplication that provides all the architecture and performance related information.

There is also some DSP implementations, such as [17] and [18], but due the lack of a good metric to evaluate different forms of showing results, as well the diverse ways to describe the Montgomery algorithm, we chose to program it in software targeting a processor and a DSP.

We used published results of an ASIC implementation [20] and a FPGA design [15]. These two examples where chosen because they use very similar versions of the algorithm presented in the Section 2. The goal of this approach is to have a way to compare one version of Montgomery algorithm implementation for different embedded platforms. In this Section a brief outline is given, whereas the results are presented in Section 5.

3.1. Software implementations

3.1.1. Generic Purpose Processor Implementation

The platform used is a Sun workstation, with a 1GHz processor and 1Gb RAM. The algorithm is coded in C with a specific library to manipulate large integers (called GiantInt [19]). This is a bit-wise level implementation of the Montgomery algorithm. The Figure 4 shows a fragment of the program to highlight the number of instructions behind a function. The algorithm is quite simple a priori, but to compute large numbers, for instance, performing an addition between two large[3] numbers, is done by calling K times a function made of K iterations, where K is the number of bits (i.e. 1024). In other words, the complexity is $O(n^2)$.

```
for (i = 0; i < K; i++) {
  if (bitval(A,i)==0) gtog(zero, AiB);
  else gtog(B, AiB);
  normal_addg(AiB,R);
  if (bitval(R,0)==0) gtog(zero, Mr);
  else gtog(M, Mr);
  normal_addg(Mr,R);
  idivg(2,R);
```

Figure 4 - Montgomery C program (fragment)

3.1.2. DSP Implementation

A Texas Instruments 32 bit 300MHz DSP (C62xx family [21]), was chosen to evaluate the performance of the Montgomery algorithm in such kind of device. The algorithm's version implemented is presented in the Figure 3, coded in C, without any use of specials libraries. Similarly to the GPP, this program is totally iterative.

3.2. Hardware designs

3.2.1. FPGA Implementation

The design preferred was the iterative version presented in [15], since the algorithm used is the same used to program the GPP version (shown in Figure 2). The detailed architecture of the Montgomery modular multiplier is given in Figure 5. It uses two muxes, two adders, two shift registers, three registers as well as a controller.

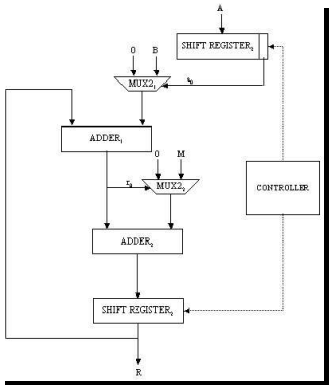

Figure 5 - Nedjah's [15] iterative modular multiplier

The first multiplexer of the proposed architecture, i.e. $MUX2_1$, passes 0 or the content of register B depending on whether bit a_0 indicates 0 or 1 respectively. The second multiplexer, i.e. $MUX2_2$ passes 0 or the content of register M depending on whether bit r_0 indicates 0 or 1 respectively. The first adder, i.e. ADDER1, delivers the sum $R + a_i \times B$, and the second adder, i.e. $ADDER_2$, yields the sum $R + M$. The shift register SHIFT $REGISTER_1$ provides the bit a_i. At each iteration i of the multiplier, this shift register performs a right-shift operation once, so that the least significant bit of SHIFT $REGISTER_1$ contains a_i.

3.2.2. ASIC Implementation

The publication [20] describes an ASIC implementing a two-stages pipelined version of Montgomery algorithm. Figure 6 depicts that the Modular Multiplication unit (MM) which can be replicated to obtain more parallelism.

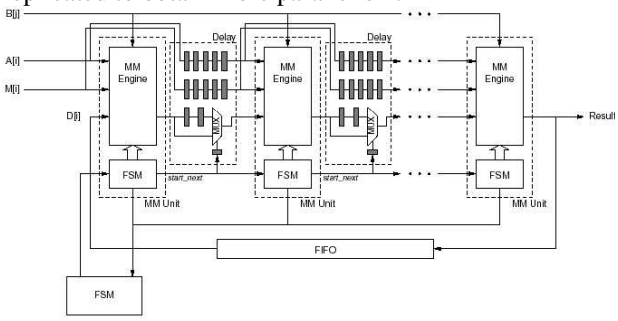

Figure 6 - ASIC for Modular Multiplication

The MM Engine is the combination of the two unified digit multipliers with a three input adder into a two stage pipelined block. This component delivers the computational power of the arithmetic unit, but it relies on control signals coming in from the outside.

Each of the two digit multipliers is followed by a register. A signal for each of these is used to control its behavior to either accept a new result from the multiplier at the time of a clock edge, or preserve its value. These two registers mark the boundary of the first pipeline stage of the engine.

In the second stage, a three input unified adder sums up the results of both multipliers of width $2w$ along with a third value, which is the feed-back of the upper $w+1$

[3] 512 bits or more

bits of the previous cycle's result shifted to the right by *w* positions.

The critical path of the second pipeline stage is shorter than that of the first stage, which includes the digit multiplier. Balancing the two stages in terms of delay would certainly be beneficial for achieving a higher clock frequency, but it is connected with a number of other problems, like additional latency during the initial computation of the parameter *R*. Another possibility might be to add a third pipeline stage by partitioning the multipliers into two balanced stages. Again this would mean increased latency during initialisation, but it would also influence the clock period in a positive way.

The circuit was designed from VHDL and synthesized with ADK from Mentor Graphics, and the technology used was AMS 0.5µ CMOS.

4. The Systolic Ring

For time consuming applications, particularly at word level such as modular exponentiation in cryptography, the use of coarse grained reconfigurable architecture suggests good improvements. In this way, this work investigates the possibility of use the Systolic Ring [7], an architecture whose Figure 7 gives an overview, and is described below*:*

- The operative layer is no longer CLB based, but use a coarse-grained granularity component: the Dnode (Data node). It is a datapath component, with an ALU and a few registers. This component is configured by a microinstruction code.
- The configuration layer follows the same principle as FPGAs, it's a RAM which contains the configuration of all the components (Dnodes and interconnect) of the operative layer.
- A custom RISC core is used, with a dedicated instruction set as configuration controller; its task is to dynamically manage the configuration of the network and also to control the data communications between the reconfigurable core and the host CPU.

This architecture is thus not intended to be a stand-alone solution, rather an accelerator for data oriented intensive computing, which would take place in a SoC. Figure 7 shows schematically the system in a SoC context. The GPP can thus confide the most demanding part of a given application to our IP core.

From a functional point of view:

- The operating system running on the host processor loads a given application, specially designed for a co-execution. The application is constituted by host-executable code (directly loaded on the host memory) and Systolic Ring configuration controller executable code (management code).
- The host processor first uploads the management code to the configuration controller memory (which has it own program memory). This object code is specially designed to dynamically manage the configuration of the network (the content of the RAM thus can be changed from one cycle to another), as to say, the functionality of the operating layer. Each clock cycle, the configuration controller is able to change up to the entire content of the RAM thanks to its dedicated instruction set.
- Once done, the core is ready to compute. The host processor sends the data to the operating layer via a specific scheme and then get back the computed data. As the configuration is dynamically managed, it is possible to multiplex the sent data, and to compute them by several sequential (hardware multiplexing) or concurrent (static) synthesized datapaths.

5. Results and Conclusions

Figure 8-I shows the operations to be implemented in the reconfigurable architectures, according to the algorithm presented in Figure 2. The first step to implement the Montgomery algorithm is to define its corresponding dataflow graph, depicted in Figure 8-II.

The dataflow graph is manually mapped to available operators in the Systolic Ring. In the dataflow graph, after each operation, the data is stored in a register (not shown in the Figure 8-II). However the register *R* is shown to outline that there is a MAC operation, where the value is re-injected. This specific dataflow graph is mapped to 7 Dnodes.

Figure 8-III shows that each Dnode performs only one operation. If more than one operation should be mapped to the same Dnode, the Systolic Ring can be reconfigured in one clock cycle. This feature must be compared to the commercial FPGAs (e.g. Virtex), where reconfiguration requires thousands of clock cycles instead only one clock cycle of the systolic ring, a net advantage of the coarse grain architectures against fine ones. In fact, in one clock cycle a pair of Dnodes linked to a switch is configured. So, eight clock cycles are required to configure the entire systolic ring.

Figure 8-III also shows that after the first iteration (4 cycles), *n* cycles are necessary to compute one modular multiplication. So, to compute a single multiplication of operands with 1024bits, 1032 cycles are required.

Even if the algorithm version implemented is not the best choice (bit level instead word level), the coarse grain architecture shown to be very competitive. The results of the implementations briefly presented in Section 3 (with 1024bits operands), and the results of SRing, are shown in Table 1.

Related with Systolic Ring, there is some works that are being done, for example, the word-level implementation of the Montgomery algorithm, as well to

test the local mode programming (it is possible to program 16 instructions at one Dnode, see [7]).

Although the preliminary results seems to be promising, some modifications to the Systolic Ring can be performed to adapt it to public key cryptographic algorithms. Modifying the Dnode by inserting specific instructions (such as modular operations) should greatly improve the performance. The idea is to modify the current ALU to include modular operators, but not changing all Dnode scheme. For the time being, we are evaluating the area impact of this changes.

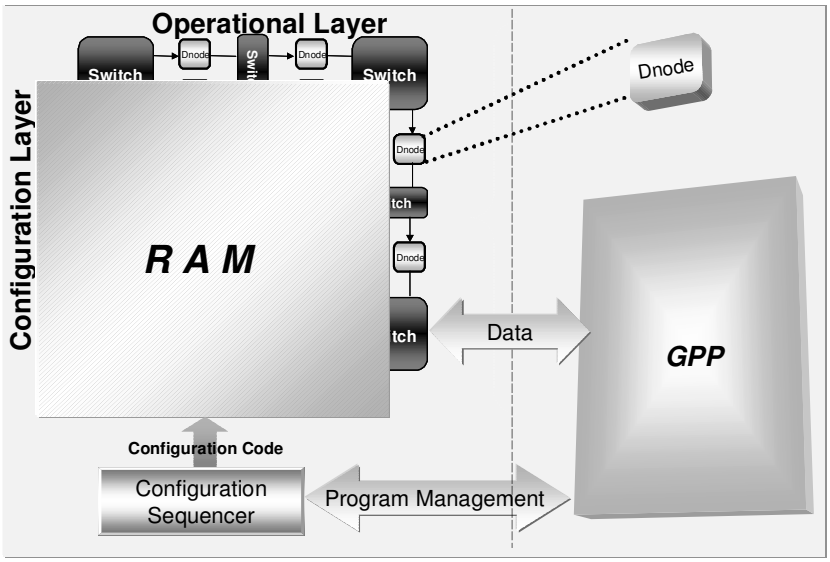

Figure 7 - Systolic Ring overview

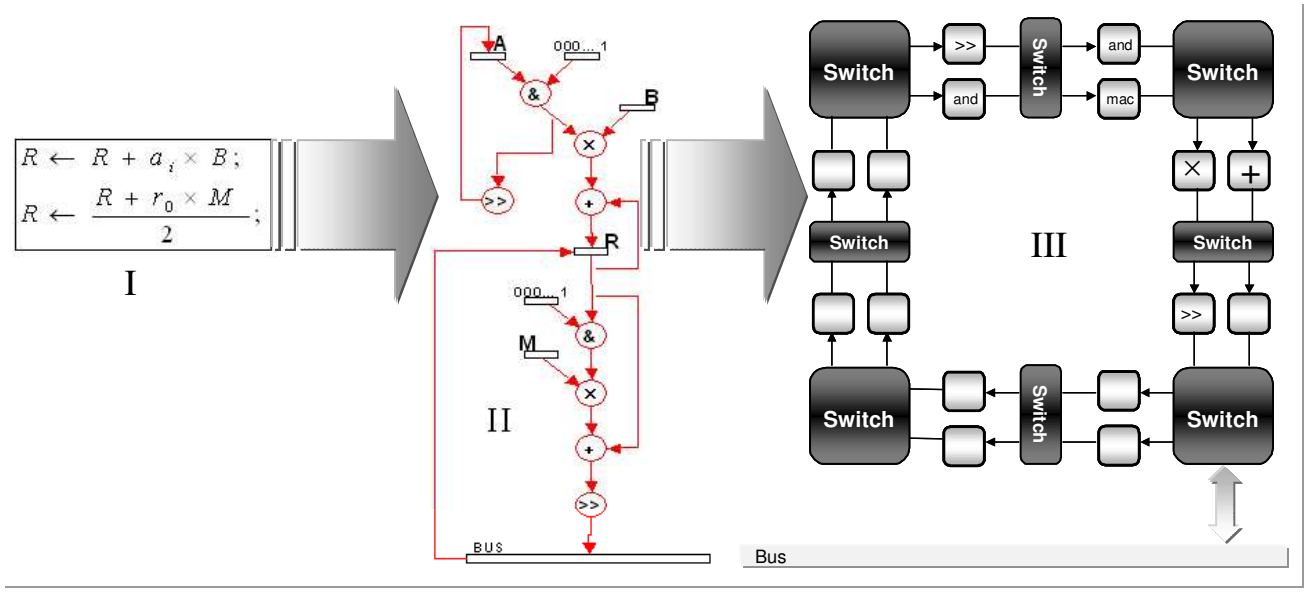

Figure 8 - Montgomery algorithm mapped to Dnodes

We are investigating the performance improvements by implementing the Montgomery algorithm in local mode of programming the SRing, and also studying the hypothesis of security enhancement of RSA cryptosystem through dynamically reconfiguring the algorithms to perform modular operations. The question is: changing from Montgomery to Barret's modular multiplication an then to other one, and so on, will make the system more robust?. But we do not have concluding results to these two hypotheses yet.

Table 1 - Comparison among different architectures

Architecture	Clock (MHz)	Word Size (Bits)	Cycles	Time to compute A×B mod M
GPP	1000	1024	2.099.200	~2,1ms
DSP	300	32	214.094	~710ms
FPGA[15]	20	1024	324	~16,2ms
ASIC[20]	33	32	1183	~35,8ms
SRing	200	1024	1032	~5,2ms

Answering the question proposed at the paper's title, although some adaptations are required to gain more performance, the coarse grain reconfigurable architecture shown in this paper is a promising platform to implement cryptographic algorithms.

6. References

[1] R. Rivest, A. Shamir and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, vol 21. pp. 120-126. 1978.

[2] V. S. Miller. "Use of Elliptic Curves in Cryptography". Advances in Cryptology – CRYPTO '85 Proceedings, Lecture notes in computer Science , Springer-Verlag. pp 417-426. 1986

[3] P. Barret. "Implementing the RSA public-key encryption algorithm on a standard digital signal processor". In A. M. Odlysko, editor: Advances in Cryptology – CRYPTO '86 Proceedings, Lecture notes in computer Science, Springer-Verlag, pp 311-323. 1987.

[4] A. D. Booth. "A signed binary multiplication technique". Quarterly Journal of Mechanics and Applied Mathematics, vol 4. pp 236-240. 1951

[5] G. R. Blakley. "A computer algorithm for the product AB modulo M". IEEE Transactions on Computers, vol 32. pp 497-500. 1983.

[6] P. L. Montgomery, "Modular Multiplication Without Trial Division", Mathematics of Computation, Vol 44. pp 519-521. 1995.

[7] G. Sassatelli, et al. "The systolic ring : A dynamically reconfigurable architecture for embedded systems". 11th International Conference on Field-Programmable Logic and Applications, FPL '01. Belfast, Northern Ireland. Pp 409-419. 2001

[8] A. Menezes, et al. "Handbook of Applied Cryptography". CRC Press. Pp 600. 1996.

[9] Ç. K. Koç et al. "Analyzing and Comparing Montgomery Multiplication Algorithms. IEEE Transactions on Microprocessors, Vol 16. Pp 26-33. 1996.

[10] T. Blum, C. Paar. "Montgomery Modular Exponentiation on Reconfigurable Hardware". 14th IEEE Symposium on Computer Arithmetic, ARITH-14 Proceedings, Australia. Pp 14-16. 1996

[11] J. Põldre et al. "Modular Exponent Realization on FPGAs" . In Field-Programmable Logic and Applications, 8th International Workshop, FPL'98, Estonia. Pp 336-347, 1998.

[12] G. Orlando and C. Paar, "A high performance elliptic curve processor for GF(2m)," in Workshop on Cryptographic Hardware and Embedded Systems - CHES 2000, vol. LNCS 1965, (Worcester, Massachusetts, USA), Springer-Verlag, 2000.

[13] K. Cho et al. "High-Speed Modular Multiplication Algorithm for RSA Cryptosystem". IECON pp. 479-483, 2001.

[14] A. Bernal et al. "Hardware for Computing Modular Multiplication Algorithm". 13th Conference on Design of Circuits and Integrated Systems (DCIS'98), Spain. 1998.

[15] N. Nedjah, L. M. Mourelle. "Two Hardware Implementations for Montgomery Modular Multiplication: Sequential versus Parallel". Proceedings of 15th Symposium on integrated circuits and system design SBCCI, Brasil. 2002.

[16] A. Daly, W. Marnane. "Efficient Architectures for implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic". Proceedings of FPGA '02, CA, USA. 2002.

[17] F. Sousa, P Felix. "The computation of Extended-Precision Modular Arithmetic on a DSP architecture". Proceedings of International Conference on Signal Processing Applications and Technology, ICSPAT '96, Massachusetts, USA. 1996.

[18] J. Guajardo, R. Bluemel, U. Krieger, C. Paar, Efficient Implementation of Elliptic Curve Cryptosystems on the TI MSP430x33x Family of Microcontrollers, In Kwangjo Kim (Ed.), Fourth International Workshop on Practice and Theory in Public Key Cryptography - PKC 2001, volume LNCS 1992, Korea, 2001.

[19] Perfect Scientific. "Library of routines for large integer arithmetic and number theory". http://www.perfsci.com/free/giantint/index.html

[20] G. Gaubatz, "Versatile Montgomery Multiplier Architectures", Master's Thesis, Worcester Polytechnic Institute, Worcester, MA, USA. May 2002.

[21] Texas Instruments. "C6000 DSPs:C62x™ DSPs". http://dspvillage.ti.com/docs/catalog/generation/overview.jhtml?templateId=5154&path=templatedata/cm/dspovw/data/c62_ovw&familyId=326

[22] R. Hartenstein. "Are we really ready for the break-through?" – invited KeyNote 10th Reconfigurable Architectures Workshop. RAW 2003. Proceedings.Nice, France. 2003.

[23] A. Gutub et al. "Scalable and Unified Hardware to Compute Montgomery Inverse in GF(p) and GF(2)". Cryptographic Hardware and Embedded Systems. CHES 2002. Redwood Shores, USA, August 13-15, 20. pp 484-495. 2002.