Carry-Save Montgomery Modular Exponentiation on Reconfigurable Hardware

A. Cilardo, A. Mazzeo, L. Romano, G. P. Saggese
Università degli Studi di Napoli Federico II, Italy
{acilardo, mazzeo, lrom, saggese}@unina.it

Abstract

In this paper we present a hardware implementation of the RSA algorithm for public-key cryptography. Basically, the RSA algorithm entails a modular exponentiation operation on large integers, which is considerably time-consuming to implement. To this end, we adopted a novel algorithm combining the Montgomery's technique and the carry-save representation of numbers. A highly modular, bit-slice based architecture has been designed for executing the algorithm in hardware. We also propose an FPGA-based implementation of the architecture developed. The characteristics of the algorithm, the regularity of the architecture, and the data-flow aware placement of the FPGA resources resulted in a considerable performance improvement, as compared to other implementations presented in the literature.

1. Introduction

In the recent years, we have witnessed to an increasing deployment of hardware devices for providing security functions, such as confidentiality, authentication, integrity and non-repudiation [1]. Among the existing techniques the Rivest-Shamir-Adleman (RSA) algorithm [2] is by far the most widely adopted public-key cryptography algorithm. The RSA algorithm has a number of applications [1], such as encryption and digital signature. The basic operation of this algorithm is modular exponentiation on large integers, i.e. $Y = X^E \bmod N$, which is used for both decryption/signature and encryption/verification. The security level of an RSA cryptosystem is tied to the length of the modulus N. A modulus of at least 768 bits is recommended, but one had best use 1024-bit moduli at least for long-term security. All operands involved in the computation of modular exponentiation have normally the same size as the modulus.

All existing techniques for computing $X^E \bmod N$ reduce modular exponentiation to a sequence of modular multiplications. Several sub-optimal algorithms have been presented in the literature to compute the sequence of multiplications leading to the Eth power of X, such as binary methods (RL-algorithm and LR-algorithm), M-ary methods, Power Tree, and more [3, 4]. We adopted the method known as Binary Right-to-Left Algorithm [4] which consists of repeated squaring and multiplication operations. This choice was motivated by its simple and efficient hardware implementation.

Since modular multiplication is the core computation of all modular exponentiation algorithms, the efficiency of its execution is crucial for any implementation of the RSA algorithm. Unfortunately, modular multiplication is a complex arithmetic operation because of the inherent cost of multiplication and modulo operations. Several techniques have been proposed in the last years for achieving efficient implementations of modular multipilcation. In particular, Blakley's method [5] and Montgomery's method [6] are the most studied ones. Indeed, they are the only algorithms suitable for practical hardware implementation [3]. Both Blakley's method and Montgomery's method perform the modular reduction during the multiplication process. No division operation is needed at any point in the process. However, Blakley's method needs a comparison between two large integers at each step of the modular multiplication process, while the Montgomery's method does not. This is achieved by resorting to a representation of the operands as a residue class modulo N. Furthermore, the Montgomery's technique requires some preprocessing and postprocessing steps, which are needed to convert the numbers to and from the residue based representation. However, the cost of these steps is negligible when many consecutive modular multiplications are to be executed, as in the case of RSA. This is the reason why the Montgomery's method is considered the most efficient algorithm for implementing RSA operations. There exist several versions of the Montgomery's algorithm, depending on the number r used as the radix for the representation of numbers. In hardware implementations r is always a power of 2.

The Montgomery's algorithm is in turn based on repeated additions on integer numbers. Since the size of operands is as large as the modulus, the addition operation

1530-1591/04 $20.00 (c) 2004 IEEE

turns out to be the most critical step from the implementation viewpoint. In this paper, we present a novel algorithm combining the Montgomery's technique with the radix-2 carry-save representation of numbers so that the basic addition has a time complexity which does not scale with the operand size.

A bit-slice based architecture has been developed which exploits the properties of the carry-save based algorithm. This architecture is highly modular, regular, and scalable with respect to the length of the modulus N. It is also well-suited for an efficient fully-parallel hardware implementation, which can be hosted in a medium size reconfigurable device.

Another major contribution of this paper is the Field-Programmable Gate Array (FPGA) implementation of the proposed architecture. We resorted to a data-flow aware placement technique to achieve an optimal result from the synthesis process, with respect to the basic bit-slice and the overall structure.

The characteristics of the algorithm, the regularity of the architecture, and the data-flow aware placement resulted in a 32% improvement in the total time for a modular exponentiation process, compared to the best implementation in the literature of the radix-2 Montgomery's algorithm.

The rest of the paper is organized as follows. Section 2 presents the proposed algorithm for computing the RSA modular exponentiation. Section 3 describes the architecture implementing the modular exponentiation algorithm. Section 4 provides details about physical implementation of the proposed architecture and presents the results achieved. Section 5 gives some comparisons with previously reported implementations of modular exponentiation. Section 6 concludes the paper with some final remarks.

2 Algorithms

For implementation of modular multiplication we exploited some optimizations of Montgomery's product first described by Walter [7]. Let modulus N be represented with K bits, and R equal to 2^{K+2}. Please note that N is always odd in the RSA algorithm. The N-residue of A with respect to R is defined as the positive integer $\bar{A} = A \cdot R \bmod N$. Montgomery Product [6] of residues of A and B, $MonProd(\bar{A}, \bar{B})$, is defined as $(\bar{A} \cdot \bar{B} \cdot R^{-1}) \bmod N$, that is the N-residue of the desired $A \cdot B \bmod N$. If $A, B < 2N$, combining [7] and [3], the following radix-2 binary add-shift algorithm can be employed to calculate $MonProd$:

Algorithm 1 - *Montgomery Product MonProd(A,B) radix-2.*
Given $N = \sum_{i=0}^{K-1} N_i \cdot 2^i$, $A = \sum_{i=0}^{K+2} A_i \cdot 2^i < 2N$, $B < 2N$, where $N_0 = 1$, A_i, $N_i \in \{0, 1\}$, A_{K+1}, A_{K+2} = 0, computes a number falling in [0, $2N$[which is modulo N congruent with desired $(A \cdot B \cdot 2^{-(K+2)}) \bmod N$

1. $U = 0$
2. For $j = 0$ to $K + 2$ do
3. if ($U_0 = 1$) then $U = U + N$
4. $U = (U/2) + A_j \cdot B$
5. end for

Algorithm 1 is the algorithm we used to implement the digit-serial implementation we presented in [11]. Starting from this algorithm, we rearranged it to accept a carry-save representation of numbers. Basically our version of the Montgomery's algorithm computes $MonProd(A, B)$ based on a modified form of the previous radix-2 Montgomery algorithm avoiding carry propagation in the loop body.

Algorithm 2 - *Carry-Save Montgomery Product MonProd(A,B) radix-2.*
Given $N = \sum_{i=0}^{K-1} N_i \cdot 2^i$, $A = \sum_{i=0}^{K+4} A_i \cdot 2^i < 2N$, $B < 2N$, where $N_0 = 1$, A_i, $N_i \in \{0, 1\}$, $A_{K+1} \ldots A_{K+4}$ = 0, computes a number falling in [0, $2N$[which is modulo N congruent with desired $(A \cdot B \cdot 2^{-(K+2)}) \bmod N$

1. $S := 0, \quad C := 0$
2. $V := 0$
3. for $h := 0$ to $K + 4$ do
4. $q := f(S_2, S_1, C_1, C_0, N_1)$
5. $V_{NEXT} := A_h \cdot 4B + q \cdot N$
6. $S := S/2, \quad (S, C) := S + C + V$
7. $V := V_{NEXT}$
8. end for
9. return $S/2 + C$

where $(S, C) := S + C + V$ denotes a carry-save addition, i.e. given $S = \sum_{i=0}^{K+3} S_i \cdot 2^i$, $C = \sum_{i=0}^{K+3} C_i \cdot 2^i$, $V = \sum_{i=0}^{K+3} V_i \cdot 2^i$, the updated values of S and C are obtained as $S_i := S_i \oplus C_i \oplus V_i$ and $C_i := S_i C_i + S_i V_i + C_i V_i$.
The value of q is evaluated as follows:

$$q = \begin{cases} S_2 \oplus C_1 & \text{when } (S_1 = 0) \wedge (C_0 = 0), \\ \overline{S_2 \oplus C_1 \oplus N_1} & \text{when } S_1 \oplus C_0 = 1, \\ \overline{S_2 \oplus C_1} & \text{when } (S_1 = 1) \wedge (C_0 = 1). \end{cases}$$

During the hth iteration, q represents the least significant bit of the partial product U ($U = S/2 + C$) computed during the $(h + 1)$th iteration. The least significant bits of the current U are needed to choose which value of V to add during the next operation in the loop of the Montgomery's algorithm. These bits can be easily derived even though numbers are in carry-save form. It is worth emphasizing that in our $MonProd$ algorithm step 6, which performs the current operation, has no dependency upon steps 4-5, which decide on the subsequent operation. Thus, control operations and data operations can be executed concurrently even if the addition is implemented on a fully parallel structure. The quantities $4B$ and $4B + N$ can be computed and stored before executing the $MonProd$ algorithm, and added during the $MonProd$ loop according to the values of A_h and q (step 5).

In the following we report the exponentiation algorithm for computing $X^E \bmod N$ known as Right-To-Left binary method [4], modified in order to take advantage of Montgomery's product as defined in Algorithm 2.

Algorithm 3 - *Right-To-Left Modular Exponentiation using Montgomery Product.*
Given $N = \sum_{i=0}^{K-1} N_i \cdot 2^i,\ X < N,\ E = \sum_{i=0}^{H-1} E_i \cdot 2^i < N$, and $W = (2^{K+2})^2 \bmod N$. computes $P = X^E \bmod N$.

1. $Z := MonProd(X, W)$
2. $P := MonProd(1, W)$
3. for $j := 0$ to $H - 1$ do
4. $\quad Z := MonProd(Z, Z)$
5. $\quad$ if $(E_j = 1)$ then $P := MonProd(P, Z)$
6. end for
7. return $MonProd(P, 1)$

where $MonProd(A, B)$ is a number falling in [0, $2N$[which is modulo N congruent with $(A \cdot B \cdot 2^{-(K+2)}) \bmod N$.

The first phase (steps 1-2) calculates residues of initial values X and 1. For a given key value, the factor $W = (2^{K+2})^2 \bmod N$ remains unchanged. It is thus possible to use a precomputed value for such a factor and reduce residue calculation to a $MonProd$. The core of the computation is a loop in which modular squares are performed, and previous partial result P is multiplied by Z, based on a test performed on the value of ith bit of E. It is worth noting that, due to the absence of dependencies between instructions 4 and 5, these can be executed in parallel. Instruction 7 allows to switch back from the residue domain to the normal representation of numbers. Note that $MonProd$ normally returns numbers in the range [0, $2N$[. However, as Walter proved in [8], the last Montgomery's product of Algorithm 3 (step 7) produces always a number less than N. Thus, instruction 7 of Algorithm 3 does not need any reduction step and the result of the exponentiation process can be directly output.

3 Architecture

A fully parallel architecture has been developed to meet the characteristics of the $MonProd$ and modular exponentiation algorithms.

In particular, the modular exponentiation Algorithm 3 consists of a sequence of $MonProd$ pairs included in steps 1-2 and 4-5. In each of these steps the first $MonProd$ operation will be referred to as Z product, while the second as P product. In each $MonProd$ occurring in the modular exponentiation algorithm, A and B will denote the left operand and the right operand, respectively. The Z product and the P product can be performed concurrently, and can share the same right operand B.

As far as the Algorithm 2 is concerned, provided that the two quantities $4B$ and $4B + N$ are pre-computed and stored before starting the $MonProd$, steps 4-5 can be accomplished just by selecting for V_{NEXT} one of the quantities $\{0, N, 4B, 4B+N\}$, depending on the current values of the least significant bits of S, C, N. The carry-save addition (step 6) does not depend on the selection done during the same iteration. In fact, the selection for its operands is performed during the previous iteration by means of V_{NEXT} and V. This allows the circuit to perform concurrently the selection and the addition operations, and thus to break the critical path of the architecture. Note that the computation of the selection bit q as reported in the previous section has the same time complexity than a carry-save addition. The $MonProd$ algorithm also requires two carry-propagate additions, i.e. $4B+N$ in the pre-processing phase and $S/2+C$ in the post-processing phase.

The architecture implementing the modular exponentiation and $MonProd$ algorithms is shown in Figure 1. It works on up to L-bit moduli and it is composed of $L + 4$ bit-slices, where $L + 4$ is the maximum size of intermediate results. The bit-slice structure is shown in Figure 2. It consists of two distinct sections performing concurrently the Z and P product.

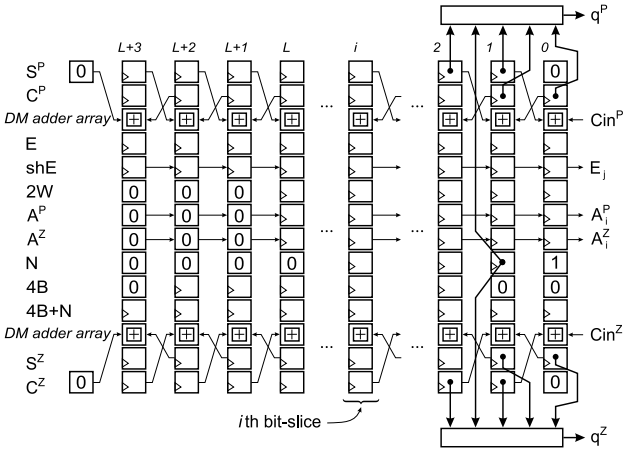

Figure 1. The overall architecture for Montgomery Modular Exponentiation.

The superscripts Z and P in the figures indicate that the corresponding signals and components are specifically involved in Z and P product computation. The flip-flops for the shared quantities N, $4B$, and $4B + N$ are accessed by both the Z and P sections.

The two blocks labelled *Dual Mode* (DM) adder in Figure 2 perform both carry-save and carry-propagate additions using the same hardware resources.

Each of the two DM adder arrays in Figure 1 uses the register C (C^Z or C^P) to hold the carry part of the carry-save pair during the loop of the $MonProd$ algorithm, while the flip-flops of register C are individually used for carry propagation during a $(K + 4)$-bit carry-propagate addition.

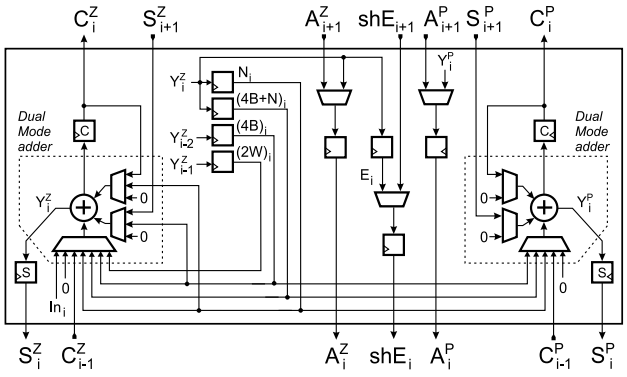

Figure 2. The structure of the ith bit-slice.

More precisely, within the ith bit-slice, each of the two DM adders can add the ith bit of $S/2$ (i.e. S_{i+1}), C_i, and one of $\{0, N_i, (4B)_i, (4B+N)_i\}$ in carry-save mode for executing $S := S/2$, $(S, C) := S + C + V$ in the $MonProd$ loop body taking one clock cycle altogether. In carry-propagate mode the DM adder can add the bits S_{i+1}, C_i together with the carry coming from the $(i-1)$th bit-slice for executing the $MonProd$ post-processing addition $S/2 + C$, taking $K + 4$ clock cycles altogether. The Z DM adder can also add the bits $(4B)_i$ and N_i in carry-propagate mode for executing the $MonProd$ pre-processing addition $4B + N$. It is worth noting that the two carry-propagate additions $S/2+C$ for the P and the Z $MonProd$ (performed concurrently by both the P section and the Z section after the $MonProd$ loop) and the carry-propagate addition $4B + N$ (where B is the previous Z product and the addition is performed before starting the following $MonProd$ loop) can overlap during the modular exponentiation process, thus taking $K+5$ clock cycles altogether. The shifts for $S/2$ and $4B$ are wired. Shift-registers are used to handle the quantities E, A^Z, and A^P.

Please note that each bit-slice communicates only with the two preceding bit-slices, that is, all data signals are strictly local. This is an advantageous condition for any hardware implementation. Furthermore, control signals can be easily broadcast since the $MonProd$ algorithm allows to pipeline controller and data path operations. In fact, the selection (steps 4-5 of Algorithm 2) and the addition (step 6) are independent.

Altogether, a complete modular exponentiation process takes $(2K+14)(H+2)+K+7$ clock cycles, where $K \leq L$ is the actual bit size of the modulus and H is the bit size of the exponent. The dominant term $2KH$ is due to $K + 5$ carry-save additions and the overlapped $(K + 4)$-bit carry-propagate additions, to be iterated $H + 2$ times within the modular exponentiation algorithm.

It is worth noting that our $MonProd$ algorithm could be modified to hold the $MonProd$ result in carry-save form throughout the modular exponentiation process. However, we did not choose this option since the change would have required more registers and given no time advantage since the $MonProd$ post-processing conversion $S/2 + C$ overlaps with the subsequent pre-processing addition $4B + N$ for the intermediate steps of the modular exponentiation process.

4. Physical Implementation

The architecture presented in the previous section has been implemented on a Field-Programmable Gate Array (FPGA). The target device belongs to the Xilinx Virtex series of FPGAs. The Xilinx Virtex architecture consists of a logic cell – or Configurable Logic Blocks (CLB) – and interconnection circuitry, tiled to form a chip. Each CLB consists of two slices, each slice containing two 4-inputs Look-Up Tables (LUTs), 2 flip-flops (FFs), and associated carry chain logic. The FPGA we used is a Virtex 2000E-8. It has 9600 CLBs, 19520 tristate buffers, and incorporates 160 fully synchronous dual-ported 4096 bit block memories named Block SelectRAM (BRAM). As far as design tools are concerned, we used Aldec Active VHDL 4.2 for describing and simulating the system, and Synplicity Synplify Pro 7.1 for synthesis, integrated in Xilinx ISE 4.1 design flow. The proposed architecture has been implemented for L = 1024, which corresponds to 1027 bit-slices to be physically placed on the chip. The system was simulated and verified, at different stages of the implementation process.

In order to exploit the regularity of the architecture and to optimize time and area performance, we adopted the following design flow:

1) *Design of the bit-slice* – we focused on the basic building block of the design, i.e. the bit-slice of Figure 2. We wrote the VHDL description of the single bit-slice and we synthesized it with Synplify. The single bit-slice requires 24 LUTs and 12 flip-flops.

2) *Placement of the bit-slice* – we launched a place-and-route process of the single bit-slice, with an area constraint of 2×3-CLB box (which is the smallest area containing 24 LUTs). We iterated this process until we found a minimum clock period of 5.25 ns for the bit-slice as a single block. Then we derived the placement constraints for each element of a bit-slice (referring to the top-left corner of the bit-slice), and embedded them in the VHDL gate-level description of the bit-slice.

3) *Placement of the data-path* – constraints were also used to place the data path structure and the controller. As we already noted, since the data signals connecting the bit-slices are local, the bit-slices should be placed according to their indexes to minimize the wire delays. Obviously, with L = 1024, 1027 bit-slices are needed and thus it is impossible to dispose them in a single row or column on a commer-

cially available FPGA. Thus, we resorted to a "serpentine" scheme to make sure each bit-slice was close to the two preceding bit-slices and to the next one. Figure 3 shows the floor-plan of the data-path and the controller on top of the target device. The bit-slices are represented by boxes which contain their indexes within the $L + 4$ cells data-path. A 84×80-CLB area was used to accommodate the complete architecture on the FPGA device. The data-path is enclosed in four regions, each one is made up of 20×13 bit-slices, requiring a 40×39-CLB area.

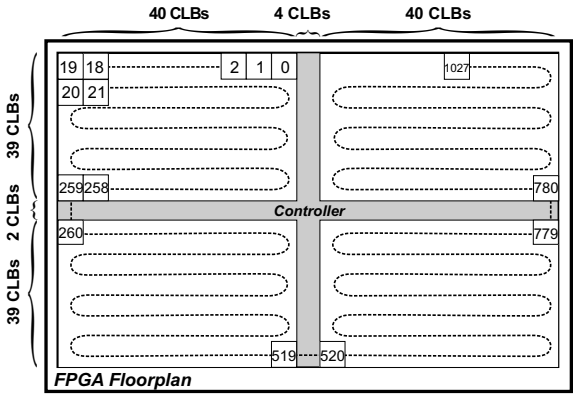

Figure 3. The data-path placement scheme.

We left a cross-shaped zone within the data-path area (which is reported as a gray area in Figure 3) to allow the synthesis tool to place the controller. This placement constraints facilitate the place-and-route step, and reduce the net delay due to control signal broadcasting, since the controller is embedded in the data-path.

Algorithm 2 has been designed to allow the controller and the data-path operations to be pipelined. In this way pipeline flip-flops can be used to break the critical path, which would run trough the controller and the data-path.

As we already noted, all controller signals have to feed the bit-slices of the data-path. This can frustrate the advantage of the locality of communications in the data-path due to remarkable net delays. To deal with the fan-out problem, flip-flops can be automatically replicated during the synthesis phase. Unfortunately the synthesizer is not able to properly "cluster" the bit-slices according to their position, so that each flip-flop would be connected with bit-slices spanned over the device with unacceptable net delays. Thus, we did not let the synthesis tool handle the replication, but we explicitly replicated the controller flip-flops in the VHDL code, in order to have a tighter control on the fan-out net, and make sure each replicated flip-flop is wired to a set of bit-slices close to each other. Figure 4 shows a detail of the floorplanner in which one of the replicated flip-flops feeds a cluster of bit-slices.

Figure 5 shows the whole design implemented on the targeted device after the place-and-route process. Altogether,

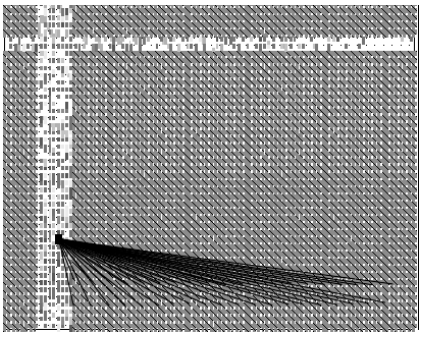

Figure 4. The wires of a replicated pipeline flip-flop of the controller.

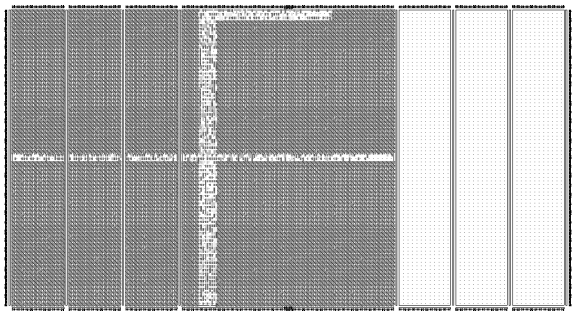

Figure 5. The implemented design.

the design takes 6,651 CLBs (24,837 LUTs and 13,500 flip-flops), i.e. it requires 69% of the target device resources. The minimum clock period is 12.88ns. From the formula of section 3, a 1024-bit full length exponentiation, i.e. a modular multiplication with an exponent E and a modulus N with a length of 1024 bits, is accomplished in 27.25 ms.

In order to prove the effectiveness of our design choices, we show some results obtained by following different approaches. In particular, we implemented an architecture based on a preliminary form of Algorithm 2, which did not permit pipelining controller and data path operations. Each case has been synthesized with and without imposing the placement constraints above explained. In Figure 6 we report the minimum clock period of the implementation for each of the four cases.

Constraints	Pipeline	Clock Period [ns]
No	No	43.23
No	Yes	28.04
Yes	No	27.75
Yes	Yes	12.88

Figure 6. Clock periods for different design choices

It is worth noting that exploiting both the design techniques (namely pipelining the controller and the data-path,

and the placement constraints), a total speed-up of about 4 times, with respect to the design without pipelining approach and with no constraints, was achieved. Please also note that each of the two design techniques roughly contributes 50% of the overall speed-up.

5. Comparison to previously reported implementations

Many different architectures have been proposed in the technical literature for modular exponentiation. However, they often rely on specific technologies and thus a fair comparison is difficult.

In [11] we proposed a serial architecture for RSA operations, with emphasis on area-performance tradeoffs. Different results were achieved depending on the value of the serialization factor, giving the designer the chance to adjust the performance of the RSA block matching resource and time constraints. The different versions of the architectures proposed in [11] were implemented on the same reconfigurable device we used for this work, so a fair comparison is possible. The fastest solution in [11] reaches an execution time of 162 ms for a 1024 bit full-length modular exponentiation, i.e. six times slower than the architecture presented in this paper. The area consumption is 1330 CLBs, i.e. five times less expensive.

To our best knowledge, the fastest FPGA-based implementation of the radix-2 Montgomery's algorithm is reported in [9, 10]. Fundamentally, [9] provides a one-row implementation of the systolic array proposed by Walter [7], in such a way that it overcomes the shortage of resources on the target device.

The implementation of the radix-2 Montgomery's algorithm proposed in [9] takes 40 ms for a 1024-bit modular exponentiation, while our implementation of the same algorithm takes 27.88 ms, achieving a 32% reduction. The physical device used in [9] is not the same as ours. This makes an exact comparison difficult in terms of hardware resource requirements and time performance. The architecture of [9] shows a critical path of a 4-bit adders. Our architecture presents a critical path of a single full-adder. Also, it should be noted that, unlike the architecture of [9], our design does not make use of optimized blocks and has considerable device-independent features.

6. Conclusions

We presented a hardware implementation of the RSA algorithm for public-key cryptography. To implement modular exponentiation we adopted a novel algorithm combining the Montgomery's technique and the carry-save representation of numbers. A highly modular, bit-slice based architecture was developed to implement the modular exponentiation algorithm in hardware, taking advantage of the properties of the carry-save representation. For each bit-slice data signals are strictly local while control signals can be delayed and easily broadcast by means of pipelined flip-flops. This was possible due to the characteristics of the adopted algorithm. We also implemented the proposed architecture on a reconfigurable device (FPGA). The design flow we followed allowed us to considerably exploit the modularity and the regularity of the architecture, achieving a 32% reduction in execution time compared to the fastest implementation of the radix-2 Montgomery's algorithm proposed in the literature on comparable hardware.

References

[1] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.

[2] R. L. Rivest et al., "A Method for Obtaining Digital Signatures", Commun. ACM, vol. 21, pp. 120-126, 1978.

[3] Ç. K. Koç, "High-speed RSA Implementation", Technical Report TR 201, RSA Laboratories, November 1994

[4] D.E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, vol. 2, Addison-Wesley, 1981.

[5] G. R. Blakley, "A computer algorithm for the product AB modulo M", IEEE Trans. on Computers, Vol.32, No.5, pp. 497-500, May 1983.

[6] P. L. Montgomery, "Modular multiplication without trial division", Math. of Computation, 44(170):519-521, April 1985.

[7] C. D. Walter, "Systolic Modular Multiplication", IEEE Trans. on Computers, Vol.42, No.3, pp. 376-378, March 1993.

[8] Walter, C. D.: 'Montgomery exponentiation needs no final subtraction', Electron. Lett., Oct. 1999, 35, (21), pp. 1831-1832.

[9] T. Blum, and C. Paar, "Montgomery Modular Exponentiation on Reconfigurable Hardware", Proc. 14th Symp. Computer Arithmetic, pp. 70-77, 1999.

[10] T. Blum, and C. Paar, "High-Radix Montgomery Modular Exponentiation on Reconfigurable Hardware", IEEE Trans. on Computers, Vol.50, No.7, pp. 759-764, July 2001.

[11] A. Mazzeo, N. Mazzocca, L. Romano, and G.P. Saggese, "FPGA-based Implementation of a Serial RSA Processor", Proceedings of the Design And Test Europe (DATE) Conference 2003, pp. 582-587.