Available online at www.sciencedirect.com
SCIENCE DIRECT®

Microprocessors and Microsystems 28 (2004) 183–191

MICROPROCESSORS AND MICROSYSTEMS

www.elsevier.com/locate/micpro

Exploring the design-space for FPGA-based implementation of RSA

A. Cilardo*, A. Mazzeo, L. Romano, G.P. Saggese

Universitá degli Studi di Napoli Federico II, Via Claudio 21, 80125 Napoli, Italy

Received 19 October 2003; revised 13 February 2004; accepted 3 March 2004

Abstract

In this paper, we present two alternative architectures for implementing the Rivest–Shamir–Adleman (RSA) algorithm on reconfigurable hardware. Both architectures are innovative, especially with respect to the implementation of modular multiplication. As to the area vs time trade-off, the two solutions are at the extremes of the design-space, since one adopts a word serial approach, while the other has a fully parallel organization. Based on the analysis of these architectures for different values of the serialization factor, we explore the design-space for the field-programmable gate array (FPGA)-based implementation of the RSA algorithm. We systematically analyze and compare the results of the two design processes with respect to two fundamental metrics, namely execution time and FPGA resource usage. We emphasize pros and cons and comment trade-offs of the two design alternatives.

Keywords: Field-programmable gate arrays; Rivest–Shamir–Adleman cryptosystem; Montgomery multiplication; Reconfigurable computing

1. Introduction

In the recent years, we have witnessed the increasing deployment of applications with a crucial need for security functions, such as confidentiality, authentication, integrity, non-repudiation, and time-stamping [1]. These include, for example, e-commerce, secure e-mail, and e-banking. Among public-key cryptography algorithms, which are fundamental for providing such security functions, the Rivest–Shamir–Adleman (RSA) algorithm [2] is by far the most widely adopted one. Unfortunately, like most cryptographic operations, the implementation of the RSA algorithm is characterized by challenging requirements, both in terms of security (i.e. tamper resistance) and performance (i.e. execution time).

To cope with such requirements, hardware implementation of RSA is usually the more convenient choice. In fact, hardware solutions generally provide a high-level of physical security and performance. Unfortunately, traditional ASIC-based implementations entail development costs which are justified only for high-volume applications. Moreover, ASIC solutions generally suffer from poor flexibility characteristics, which do not enable modification of the implemented algorithm and its parameters, such as the RSA key size. Reconfigurable hardware devices such as field-programmable gate arrays (FPGAs) are a promising solution for hardware implementation of RSA and of most cryptographic primitives in general [3]. Although not as fast as ASICs, FPGAs retain the physical security of traditional hardware solutions while providing high flexibility. Moreover, the time and costs for developing an FPGA-based solution are much lower than for an ASIC implementation.

In this paper, we present two different hardware architectures which implement RSA cryptography using FPGA technology. The first architecture exploits a word serial approach to break the basic arithmetic operations of modular exponentiation on large integers into single word operations. The flexibility of the technology allows us to use the serialization factor as a design parameter which can be adjusted to satisfy different area and time requirements. The second architecture exploits a carry-save representation of numbers to implement carry-free arithmetic operations and a fully parallel implementation of the modular multiplication algorithm. Evaluating the serial and parallel architecture for different values of the serialization factor allows us to wholly explore the design-space for the implementation of the RSA algorithm in FPGA technology.

* Corresponding author. Tel.: +39-81-768-3821; fax: +39-81-768-3816.
E-mail address: acilardo@unina.it (A. Cilardo).

0141-9331/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2004.03.009

We thoroughly analyze the two design strategies with respect to two fundamental metrics, namely the execution time and FPGA resource usage. We emphasize pros and cons of individual alternatives and discuss the main trade-offs.

The rest of the paper is organized as follows. Section 2 briefly explains the basic algorithms used to carry out RSA operations. Sections 3 and 4 describe the word serial architecture and the fully parallel architecture, respectively. Section 5 provides details of the physical FPGA-based implementation of the proposed architectures. Section 6 defines two fundamental efficiency metrics and uses them to compare the implementation results of the two designs. Section 7 concludes the paper with some final remarks.

2. Key-activities for RSA algorithm

The basic operation of the Rivest–Shamir–Adleman (RSA) cryptosystem is modular exponentiation on large integers, i.e. $Y = X^E \bmod N$, which is used for both decryption/signature and encryption/verification. The interested reader can find a thorough description of RSA cryptography in Ref. [1]. All existing techniques for computing $X^E \bmod N$ reduce modular exponentiation to a sequence of modular multiplications [4,5]. We thus focus on implementation of modular multiplication as it is the crucial activity in the execution of the RSA algorithm. The most studied—and practically implementable—algorithms are Blakley's method [6] and Montgomery's method [7]. Both of them perform the modular reduction during the multiplication process avoiding any division operation. However, it is widely recognized that the Montgomery's algorithm is the most efficient method for hardware implementation, since it does not contain any comparison on large integers [4,8], although it works on residue classes modulo N and thus needs to perform some pre-processing and post-processing computation on operands and final results. This overhead is negligible when the Montgomery's algorithm is used iteratively for computing modular exponentiation. The basic form of the Montgomery's algorithm is given as follows (we consider the radix-2 form of the algorithm [4,7]).

Algorithm 1. Given $A = \sum_{i=0}^{K-1} A_i 2^i$, $B = \sum_{i=0}^{K-1} B_i 2^i$, $N = \sum_{i=0}^{K-1} N_i 2^i$, where A_i, B_i, $N_i \in \{0,1\}$, $N_0 = 1$, $A, B < N$, computes $U = (AB2^{-K}) \bmod N$.

```
1. $U := 0$
2. for $j := 0$ to $K - 1$ do
3.    $U := U + A_j B$
4.    if ($U_0 = 1$) then $U := U + N$
5.    $U := (U/2)$
6.    end for
7. If $U > N$ then $U := U - N$
```

Note that $N_0 = 1$ is a necessary pre-condition for this algorithm which is always true in RSA cryptography, since N is the product of some prime integers.

The Right-to-Left Binary algorithm [4,5] used to compute modular exponentiation by means of the Montgomery's product $MonPro(A, B)$ is given below.

Algorithm 2. Given X, N, and $E = \sum_{i=0}^{H-1} E_i 2^i$, $E_i \in \{0,1\}$, $R = 2^K \bmod N$, computes $P = X^E \mathit{mod}\, N$.

```
1. $Z^{(0)} := MonProd(X, R^2 \bmod N)$
2. $P^{(0)} := MonProd(1, R^2 \bmod N)$
3. for $i := 0$ to $H - 1$ do
4.    $Z^{(i+1)} := MonProd(Z^{(i)}, Z^{(i)})$
5.    if ($E_i = 1$) then $P^{(i+1)} := MonProd(P^{(i)}, Z^{(i)})$
      else $P^{(i+1)} := P^{(i)}$
6.    end for
7. $P := MonProd(P^{(H)}, 1)$
```

For a given value of the key, the factor $R^2 \bmod N = 2^{2K} \bmod N$ remains unchanged. It is thus possible to use a pre-computed value for such a factor and to reduce residue calculation to a *MonPro* (steps 1 and 2). It is worth noting that, since steps 4 and 5 are independent of each other, they can be executed in parallel. This is true also for steps 1 and 2.

3. Serial architecture

In this section, we present our first proposal, i.e. a microcontrolled architecture which implements the modular exponentiation algorithm introduced in Section 2. This architecture is based on a word serial approach which breaks the basic arithmetic operations on large integers into single word operations. The word width, i.e. the serialization factor, is taken as a parameter. The architecture of this section is an improved version of the work presented by the authors in Ref. [9]. A detailed description of the architecture is given in Ref. [10].

For executing modular multiplication we used a modified version of Algorithm 1, which is given below.

Algorithm 3. Given $A = \sum_{i=0}^{K} A_i 2^i$, $B = \sum_{i=0}^{K} B_i 2^i$, $N = \sum_{i=0}^{K-1} N_i 2^i$, where $N_0 = 1$, $0 < A, B < 2N$, computes $U \in \{(AB(2^{K+2})^{-1} \bmod N), (AB(2^{K+2})^{-1} \bmod N + N)\}$.

```
1. $U := 0$
2. for $i := 0$ to $K + 2$ do
3.    if ($U_0 = 0$) and ($A_i = 0$) then $V := 0$
4.    if ($U_0 = 1$) and ($A_i = 0$) then $V := N$
5.    if ($U_0 = 0$) and ($A_i = 1$) then $V := 2B$
6.    if ($U_0 = 1$) and ($A_i = 1$) then $V := 2B + N$
7.    $U := U + V$
8.    $U := U/2$
9.    end for
```

The algorithm contains the following optimizations [9,11–14]:

- The pre-condition on the operands A and B is that they be less than $2N$ instead of N, so that no reduction step is needed (step 6 of Algorithm 1) when consecutive multiplications are performed; this modification does not affect the overall result, provided that two further iterations are added to the loop of the algorithm.
- The second operand B is shifted up by 1 bit; this speeds up the evaluation of the condition in step 3 of Algorithm 1, since U_0, and thus the next operation to be accomplished, does no longer depend on B_0 (see steps 3–6 of Algorithm 3); to adjust the value of the result, an additional iteration is added.

Fig. 1 shows the serial architecture which implements modular exponentiation based on Algorithm 3. The architecture is made of a dual-port memory section and two similar units which can execute concurrently the Montgomery's product of Algorithm 3. All blocks in the figure have a data width of S bits. Fundamentally, the operation accomplished by both P- and Z-unit is $U \leftarrow U/2 + V$, where $V \in \{0, N, 2B, 2B + N\}$, that is the loop body of Algorithm 3 (steps 3–8). This operation is broken into S-bit word operations by means of multi-precision arithmetic logic, and are executed in a pipelined fashion.

P and Z units (see Fig. 1) are used to compute concurrently the P- and Z-multiplications of Algorithm 2 (steps 1, 2, 4 and 5). In fact, the two multiplications have no data dependencies and can be parallelized. Also, they have always the same left operand (R^2 mod N in steps 1 and 2 and Z_i in steps 4 and 5) so that a unique physical location is necessary for storing it. In other words, parallelizing the two operations results in less than doubling the need for hardware resources, while increasing performance by a factor of 2. The quantities $2B$ and $2B + N$—where B is the left operand of the two multiplications—are constant during each multiplication and can be pre-computed and stored in the memory section; this speeds up the operations within the central loop of the Montgomery's algorithm. Note that an $L + 3$-bit memory location is needed to store the quantity $2B + N$ and all intermediate results, where L is the maximum size allowed for the modulus. Memory section and *RegisterU* have thus a size of $L + 3$ bits.

We briefly list the function of each component of Fig. 1: *RegisterU* can operate as an ordinary register or a multiprecision right-shift register; *Left-Shifter* can operate as a pass-through combinatorial block, or a multiprecision left-shift register used to compute the quantity $2B$; *Adder* is a multiprecision carry-propagate S-bit adder; *Adder-Register* is an ordinary S-bit register; *Register E* and *Register A* are right-shift registers used to get the ith bit of E and A during the modular exponentiation and Montgomery multiplication algorithm, respectively. Further details can be found in Ref. [10].

4. Fully parallel architecture

In this section, we present our second proposal, i.e. the fully parallel architecture. The solution presented in Section 3 serializes arithmetic operations on large integers, and enables proportioning the design with respect to the serialization factor S obtaining different area–time trade-offs. However, there exists an upper bound to the value of S for which the lower amount of clock cycles due to high serialization factors is counterbalanced by worse circuital delays. In fact, the architecture of Section 3 is based on a carry-propagate adder whose combinatorial delay becomes unacceptable for high values of S. The actual critical bound depends on the specific technology used for the physical implementation, and also on the availability of optimized logic for carry propagation on the target device. These special purpose hardware resources rarely enable carry chains longer than 256 bits. Consequently, a fully parallel architecture cannot be simply obtained as a special case of the word serial architecture corresponding to S equal to

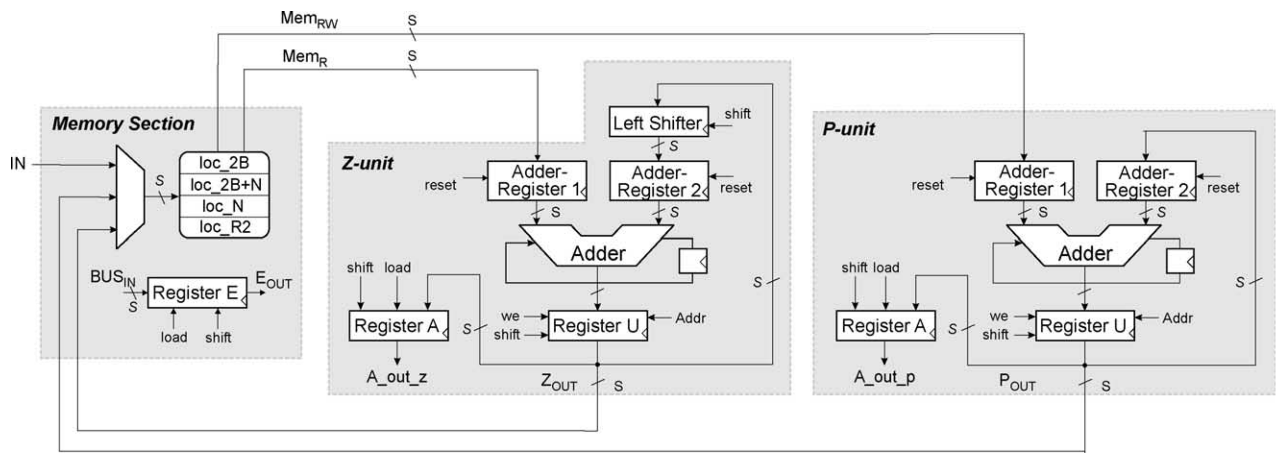

Fig. 1. Serial architecture data path.

the operand size. We thus introduce an alternative architecture based on the carry-save representation of numbers enabling carry-propagation free additions and a full length implementation of the operations required by the modular multiplication algorithm. To this aim, we introduce a modified, carry-save based version of the Montgomery's algorithm, which is given below.

Algorithm 4. Given $A = \sum_{i=0}^{K} A_i 2^i$, $B = \sum_{i=0}^{K} B_i 2^i$, $N = \sum_{i=0}^{K-1} N_i 2^i$, where $N_0 = 1$, $0 < A, B < 2N$, computes $U \in \{(AB(2^{K+2})^{-1} \bmod N), (AB(2^{K+2})^{-1} \bmod N + N)\}$.

```
1. $S := 0$, $C := 0$
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.    $V_{NEXT} := A_h 4B + qN$
6.    $S := S/2$, $(S, C) := S + C + V$
7.    $V := V_{NEXT}$
8.    end for
9. return $S/2 + C$
```

$(S, C) := S + C + V$ denotes a carry-save addition, i.e. given $S = \sum_{i=0}^{K+3} S_i 2^i$, $C = \sum_{i=0}^{K+3} C_i 2^i$, $V = \sum_{i=0}^{K+3} V_i 2^i$, 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:

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

Before explaining the architecture structure, we emphasize some aspects of Algorithm 4 which have been fruitfully exploited.

- 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 and it is needed to choose which value of $V_{NEXT} \in \{0, N, 4B, 4B + N\}$ to add during the next operation in the loop of the Montgomery's algorithm. Thus, the carry-save addition (step 6) does not depend on the selection done during the same iteration (steps 4 and 5). 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 quantities $4B$ and $4B + N$ can be computed and stored before executing the *MonPro* algorithm, and added during the *MonPro* loop according to the values of A_h and q (step 5).
- The *MonPro* algorithm requires two carry-propagate additions, i.e. $4B + N$ in the pre-processing phase and $S/2 + C$ in the post-processing phase. However, these additions occur only once for each Montgomery product and can be implemented in a pipelined fashion breaking the carry-propagation for every single bit by means of flip-flops. Also, since several consecutive multiplication pairs are to be performed, and the result of one of the two current multiplications ($S/2 + C$) is exactly the left-operand of both the following ones (B), the two $K + 4$-bit carry-propagate additions $S/2 + C$ and $4B + N$ can be performed bit-by-bit in an overlapped fashion taking only $K + 5$ clock cycles altogether.

The architecture implementing modular exponentiation by means of *MonPro* algorithms is shown in Fig. 2. As the serial architecture of Section 3, the fully parallel architecture is made of two independent sections computing concurrently the P and Z multiplications of Algorithm 2 (steps 1, 2, 4 and 5). The superscripts Z and P in the figures indicate that

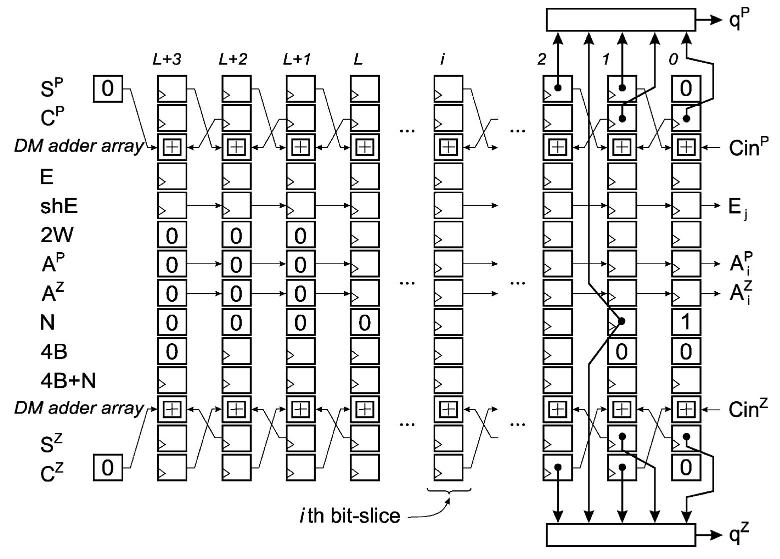

Fig. 2. Overall fully parallel architecture.

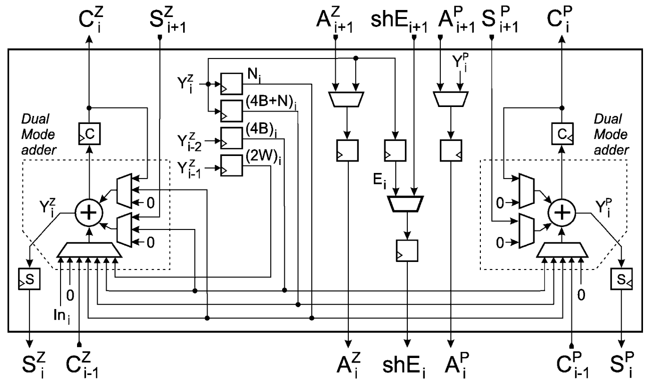

Fig. 3. Bit-slice for the fully parallel architecture.

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 architecture 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 Fig. 3. It consists of two distinct sections performing concurrently the Z and P product.

The two blocks labelled *Dual Mode* (DM) adder in Fig. 3 perform both carry-save and carry-propagate additions using the same hardware resources. Each of the two DM adder arrays in Fig. 2 uses the register C (C^Z or C^P) to hold the carry part of the carry-save pair during the loop of the *MonPro* algorithm, while the flip-flops of register C are individually used for carry propagation during a $(K + 4)$-bit carry-propagate addition. 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 *MonPro* 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 *MonPro* 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 *MonPro* pre-processing addition $4B + N$; in each clock cycle, the Z DM adder uses the bit of $S/2 + C$ just calculated in the previous clock cycle to compute the corresponding bit of $4B + N$, where $B = S/2 + C$. So, the two carry-propagate $K + 4$-bit additions $S/2 + C$ and $4B + N$ take only $K + 5$ 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 *MonPro* algorithm allows to pipeline controller and data path operations. In fact, the selection (steps 4 and 5 of Algorithm 4) and the addition (step 6) are independent.

A complete description of the architecture operation is provided in Ref. [17,10].

It should be emphasized that our solution performs a conversion operation on carry-save results at the end of each modular multiplication step. An alternative approach could keep numbers in redundant form throughout the overall modular exponentiation process. This would entail handling the quantities $2B$ and $2B + N$ in the form of carry-save pairs, and use two cascaded adders to perform the for loop of Algorithm 4. On the other hand, the execution time would approximately decrease by a factor of 2. This approach is taken, for example, in Ref. [15].

5. FPGA-based implementation

The two architectures presented in Sections 3 and 4 have been implemented on a field-programmable gate array (FPGA) device. The target device belongs to the Xilinx Virtex series of FPGAs. The Xilinx Virtex architecture consists of programmable logic cells—or Configurable logic blocks (CLBs)—and interconnection circuitry, tiled to form a chip. Each CLB consists of two slices, each slice containing two 4-inputs look-up tables (LUTs), two flip-flops (FFs), and optimized carry chain logic. Each LUT can either be used as a 16×1 bit RAM, or as a 1–16 cycle delay shift register. For our work, we used a Virtex 2000E-8, which is comprised of 9600 CLBs, 19,520 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.

5.1. Implementation of the word serial architecture

We implemented the serial architecture on the FPGA device for four values of the data width: $S = 32$, 64, 128, 256. The maximum size for the modulus is 1024 bits. The specific low-level implementation of each component of the architecture was designed in order to take advantage of the resources provided by the FPGA. For example, shift registers found a very area-effective implementation thanks to LUTs used as 16-bit shift registers. On the other hand, LUTs used as dual-port memory elements were particularly suitable for implementing wide registers, such as *RegisterU* and the memory section: in fact, these registers store internal data of $K + 3$ bits, but read and write data of S bits, making it profitable to use memory elements instead of multiplexers and flip-flop based registers. The optimal implementation is achieved when the capacity of a single memory element, that is 16 bits, is fully exploited. This happens for $S = 32$, 64. In any case, the physical implementation of the architecture is particularly area-efficient.

5.2. Implementation of the fully parallel architecture

The fully parallel solution has been implemented for 1024-bit moduli. This corresponds to 1027 bit-slices to be physically placed on the chip. In order to exploit the regularity of the architecture and to optimize time and area performance, we began the implementation process with the synthesis of the basic building block of the system, i.e. the bit-slice of Fig. 3. The result was a small regular block using 24 LUTs and 12 flip-flops and covering a 2×3-CLB rectangular box, which was well-suited for placing a modular structure on the FPGA floor plan. Then, we built the overall architecture by replicating and placing the single bit-slices. As we remarked in Section 4, 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. Since it was impossible to dispose 1027 bit-slices in a single row or column on the physical device, 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. Fig. 4 shows the floor plan of the data-path and the controller on top of the target device. The bit-slices are represented as boxes which contain their indexes within the $L + 4$ cells data path. A 84×80-slice 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.

We left a cross-shaped zone within the data-path area (which is reported as a gray area in Fig. 4) to let the synthesis tool place the controller logic. 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. Note that Algorithm 4 allows controller and data path operations to be performed in a pipelined fashion since the decision on the current operation is independent of the current data path computation. From the circuit implementation viewpoint, this means that control signals are delayed by flip-flops and can be easily broadcast over the whole architecture without introducing significant net delay.

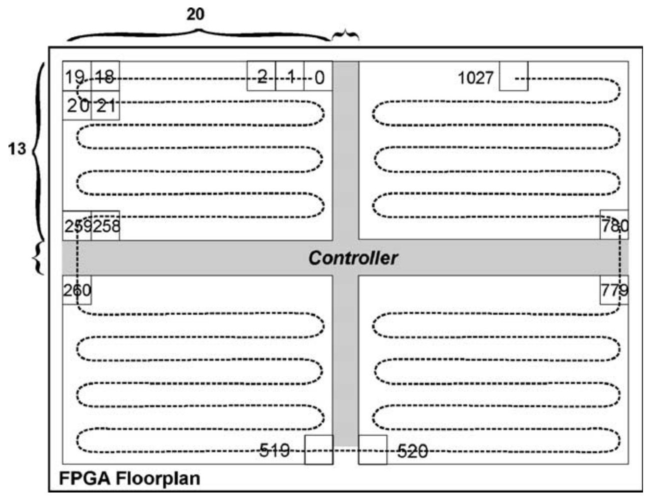

Fig. 4. Placement scheme for the implementation of the fully parallel architecture.

6. Comparison of the two approaches

Many of the architectural choices taken in the two proposed solutions have a direct impact on the implementation results since different levels of the architectural parallelism entail different ways of exploiting the specific hardware resources provided by the FPGA device. As a consequence, comparing the implementation of the two designs is not as trivial as a simple area–time trade-off depending on various choices for the serialization factor.

To give a quantitative evaluation of the achieved results and a precise comparison of the two designs, we first define a set of convenient metrics:

- the time T used to complete a full-length exponentiation, that is, an operation where the modulus, the base and the exponent consists of 1024 bits;
- the hardware resource usage. At this aim, one should distinctly take into consideration the usage of LUTs, flip-flops, BRAM blocks, and buffers tristate. However, an important figure should be not only how many resources are used of each type, but also how much balanced is their use, in order to provide an optimal exploitation of the resources usually available on an FPGA device.

As far as the execution time of the word serial architecture is concerned, let S be the architecture word length, K the modulus size, $M = \lceil (K + 3)/S \rceil$, and H the exponent size. Consider that a single multiprecision addition accomplished by a processing unit takes M clock cycles, and a Montgomery's product requires $K + 3$ additions. Also, according to Algorithm 2, a modular exponentiation process requires $H + 2$ pais of Montgomery's products. Altogether, computing a modular exponentiation process takes

$(KM + 6M)(H + 2) + 4M - (H \div S)$ clock cycles.

For the fully parallel solution, we have that a modular exponentiation process takes

$(2K + 14)(H + 2) + K + 7$ clock cycles.

Here, 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. For the exact evaluation of these formulae, refer to Ref. [10]. To determine the execution time, we used the values of the clock periods provided by the synthesis tool after a complete place-and-route process. The clock periods are of 11.16, 10.42, 13.72, and 27.03 ns for the word serial architecture and $S = 32$, 64, 128, and 256, respectively, and of 12.88 ns for the fully parallel

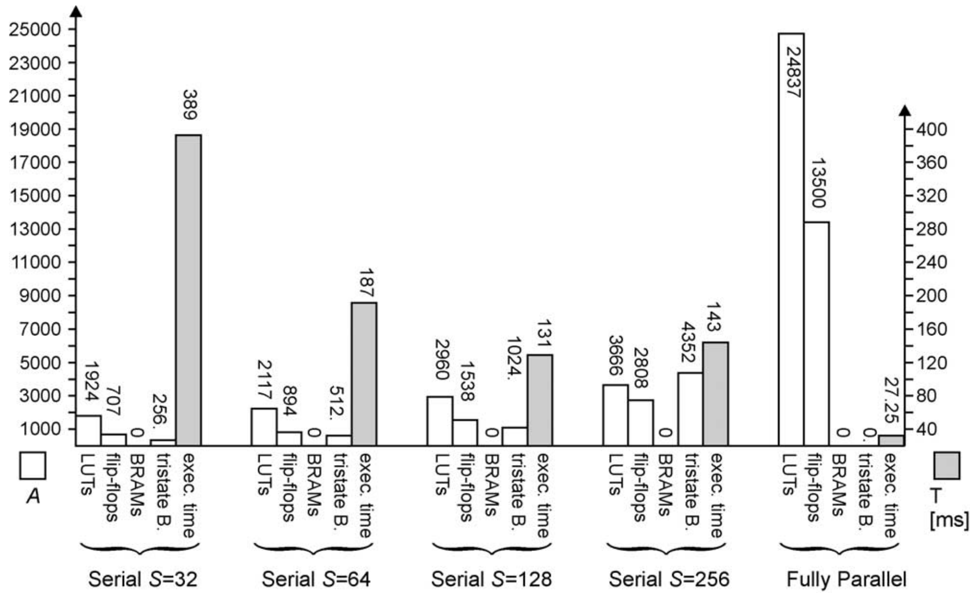

Fig. 5. Graphical representation and comparison of the achieved results.

solution. Note that the clock period for $S = 32$ is higher than $S = 64$ since for these solutions the critical path is determined by memory elements and multiplexers enabling the word serial operation, rather than by the adder. The usage of hardware resources was also derived from the reports produced by the synthesis tool. All results are graphically depicted in Fig. 5 for an immediate comparison. The values of the area–time product are shown in Fig. 6.

Note that the fully parallel solution does not use BRAM blocks and tristate buffers at all. Also, the word serial solution does not use BRAM blocks. Thus, we can use only LUTs and flip-flops for the comparison.

It should be emphasized that the presented architectures perform the fundamental RSA primitive, i.e. modular exponentiation. Thus, they are both well-suited for implementing the RSA decryption based on the Chinese remainder theorem (CRT) [16]. Such technique allows a K-bit modular exponentiation operation to be decomposed into two independent $K/2$-bit modular exponentiation processes, provided that the two prime factors of the modulus are known (and this assumption can be made for the case of RSA decryption, that is the most time consuming operation). An implementation of the CRT technique can resort to two separate processors for the two $K/2$-bit exponentiations, thereby taking roughly the same area than a K-bit implementation and executing four time faster. The use of the CRT is equally feasible for any implementation of modular exponentiation, so we did not considered it in our analysis. Nevertheless, we also implemented a CRT-based 1024-bit decryption processor based on our fastest solution, i.e. the fully parallel architecture. Such implementation takes 25,342 LUTs, 14,338 flip-flops, and execute a 1024-bit decryption in 6.9 ms.

The results shown in Figs. 5 and 6 suggest some interesting considerations, presented in the following.

- The word serial architecture makes a more efficient use of the specific hardware resources provided by the FPGA, i.e. the product AT, where A represents the number of LUTs (or flip-flops, as well) is generally better than the fully parallel architecture. In particular, most registers of the serial architecture can be implemented as memory blocks rather than wide flip-flop arrays (see Section 5) resulting in an area-effective implementation

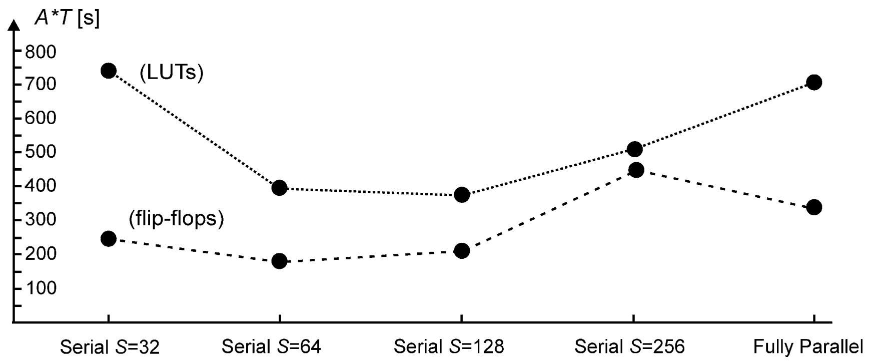

Fig. 6. Area–time products of the implemented solutions.

on the FPGA device. This is not possible for the fully parallel architecture, which basically consists of an array of customized blocks, i.e. the bit-slices. The fundamental reason of this difference lies in the level of 'granularity' of the basic arithmetic operation: the serial solution works on S-bit numbers, while the fully parallel one works on single bits and cannot take advantage of memory elements, since all bits of the operands are needed at once; also it cannot exploit optimized logic for carry-propagate addition, as it is based on a carry-save technique.

- The usage of the different types of hardware resources is rather unbalanced for the fully parallel architecture (BRAM blocks and carry chains are not exploited at all). This prevents an implemented system from exploiting all available resources on the devices, since only 'general purpose' hardware—LUTs and flip-flops—is actually used. Indeed, in certain circumstances this could be an advantageous characteristic, for example, when it is important to keep the architecture technology-independent and implementable on different devices.
- The word serial solution is not completely scalable. In fact, the results of the serial architecture do not scale linearly with S. Actually, the implementation of most of its blocks needs to be slightly adjusted with respect to S. In particular, the data width S of the word serial solution cannot be increased at will, since the maximum length of the optimized carry-propagate adder is limited and tied to the specific device. For example, the FPGA we used provides such structures with length of up to 160 bits, so that only architectures up to 128-bit can benefit from the optimized carry chains. That is the reason why we found a considerable increase in clock period for $S = 256$ (13.72 ns for $S = 128$ vs 27.03 ns for $S = 256$), and even in the absolute execution time, as it is shown in Fig. 5.
- The fully parallel architecture is much more scalable with respect to modulus size thanks to its intrinsic modularity, which enables virtually any operand length without affecting the implementation of the architecture and even without changing the controller. This would permit even to increase the operand length 'on the fly'—whether the FPGA device provides dynamic reconfigurability—by adding further bit-slices and changing the initialization values for the controller counters. Conversely, slight adjustments would be needed to re-implement the serial architecture for higher values of the modulus, although it is easy to extend the implemented design to any standard size of the modulus (2048, 4096 bits).
- Although the word serial architecture is intrinsically slower, relatively poor time performance is not always a disadvantage when associated to limited resource usage. For example, if independent RSA operations are to be performed at a high throughput, the low-physical resource requirements enable replicating the architecture within the same FPGA to allow several exponentiations to be performed in parallel and the throughput-space trade-off to be variously proportioned.

To summarize, we found that the serial approach is more effective and practical, and shows better features which enable exploiting the characteristics of modern FPGAs. The fully parallel solution is more technology-independent and could be considered, for example, for fast ASIC-based implementations.

7. Conclusions

We presented two alternative architectures for implementing RSA algorithm on re-configurable hardware. The first architecture exploited a word serial approach to break the basic arithmetic operations of modular exponentiation on large integers into single word operations. The second architecture exploited a carry-save representation of numbers to implement carry-free arithmetic operations and a fully parallel implementation of the modular multiplication algorithm. Evaluating the serial and parallel architecture for different values of the serialization factor allowed us to wholly explore the design-space for the implementation of the RSA algorithm in FPGA technology. We thoroughly analyzed the two design strategies with respect to two fundamental metrics, namely the execution time and FPGA resource usage and emphasized pros and cons of individual alternatives discussing the main trade-offs.

Acknowledgements

This work was partially funded by the University of Naples Federico II, the Consorzio Interuniversitario Nazionale per l'Informatica (CINI), the National Research Council (CNR), the Ministry of University and Research (MIUR) within the framework of SP1 project 'Sicurezza dei documenti elettronici' and FIRB project 'Middleware for advanced services over large-scale, wired-wireless distributed systems' (WEB-MINDS), and the Regione Campania within the framework 'Telemedicina' project.

References

[1] A. Menezes, P. van Oorschot, S. Vanstone, Handbook of Applied Cryptography, CRC Press, Boca Raton, FL., USA 1996.
[2] R.L. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signature and public-key cryptosystems, Commun. ACM 21 (1978) 120–126.
[3] A.J. Elbirt, W. Yip, B. Chetwynd, C. Paar, An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists, IEEE Trans. VLSI Syst. 9 (2001) 545–556.
[4] Ç.K. Koç, High-speed RSA implementation, Technical Report TR 201, RSA Laboratories, 1994.
[5] D.E. Knuth, The Art of Computer Programming: Seminumerical Algorithms, vol. 2, Addison-Wesley, Reading, MA, USA 1981.

[6] G.R. Blakley, A computer algorithm for calculating the product AB modulo M, IEEE Trans. Comput. 32 (1983) 497–500.

[7] P.L. Montgomery, Modular multiplication without trial division, Math. Comput. 44 (1985) 519–521.

[8] Ç.K. Koç, RSA hardware implementation, Technical Report TR801, RSA Laboratories, 1995.

[9] A. Mazzeo, N. Mazzocca, L. Romano, G.P. Saggese, FPGA-based implementation of a serial RSA processor, in: Proceedings of the Design and Test Europe (DATE) Conference, 2003, pp. 582–587.

[10] A. Cilardo, Modular exponentiation on reconfigurable hardware, Master Thesis, 2003, available at http://cds.unina.it/~acilardo.

[11] S.E. Eldridge, C.D. Walter, Hardware implementation of Montgomery's modular multiplication algorithm, IEEE Trans. Comput. 42 (1993) 693–699.

[12] T. Blum, C. Paar, Montgomery modular exponentiation on reconfigurable hardware, in: Proceedings of 14th Symposium on Computer Arithmetic, 1999, pp. 70–77.

[13] C.D. Walter, Montgomery exponentiation needs no final subtraction, IEE Electron. Lett. 35 (1999) 1831–1832.

[14] A. Daly, W. Marnane, Efficient architectures for implementing Montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic, in: Proceedings of the 10th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 2002, pp. 40–49.

[15] C. McIvor, M. McLoone, J. McCanny, A. Daly, W. Marnane, Fast Montgomery modular multiplication and RSA cryptographic processor architectures, in: Proceedings of Asilomar Conference on Signals, Systems and Computers, 2003.

[16] J. Quisquarter, C. Couvreur, Fast decipherment algorithm for RSA public-key cryptosystem, Electron. Lett. 18 (1982) 905–907.

[17] A. Cilardo, G.P. Saggese, A. Mazzeo, L. Romano. Carry-Save Montgomery Modular Exponentiation on Reconfigurable Hardware, in IEEE Proc. of Design, Automation and Test in Europe Conference, 2004 (DATE04), vol. 3, pp. 206–211, February 2004.