Available online at www.sciencedirect.com
ScienceDirect
Journal of Systems Architecture 53 (2007) 99–108
JOURNAL OF SYSTEMS ARCHITECTURE
www.elsevier.com/locate/sysarc

Fast hardware for modular exponentiation with efficient exponent pre-processing

Nadia Nedjah [a,*,1], Luiza de Macedo Mourelle [b,1]

[a] Department of Electronics Engineering and Telecommunication, Faculty of Engineering, State University of Rio de Janeiro, Brazil
[b] Department of Systems Engineering and Computation, Faculty of Engineering, State University of Rio de Janeiro, Brazil

Received 2 May 2006; received in revised form 29 August 2006; accepted 15 September 2006
Available online 7 November 2006

Abstract

Modular exponentiation is an important operation in several public-key cryptosystems. It is performed using successive modular multiplications. For the sake of efficiency, one needs to reduce the total number of required modular multiplications. In this paper, we propose two efficient hardware implementations for computing modular exponentiations, which are based on the *m*-ary method. The *m*-ary method includes a power pre-computation step. The first design ignores the exponent and pre-computes all possible powers while the second takes advantage of the formation of the exponent to compute only those powers that are really necessary for the rest of the computation. As it can be expected, the implementation of the first architecture, compared with that of the second one, requires more time to complete an exponentiation. In compensation, however, the first should require less hardware area than the second. We provide a comparison of these two implementations using the performance factor, which takes into account both space and time requirements.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Cryptography; Modular exponentiation; *m*-ary method

1. Introduction

Public-key cryptographic systems, such as the RSA [1,2] often involve raising large elements of some group fields to a large power. The performance of such cryptosystems is primarily determined by the implementation efficiency of modular exponentiation.

The RSA cryptosystem [1] consists of a set of three items: a modulus M and two integers D and E called *private* and *public* key, respectively. These keys must satisfy the property $T^{DE} = T \bmod M$ with plain text T obeying $0 \leqslant T < M$. Messages are encrypted using the public key as $C = T^D \bmod M$ and decrypted as $T = C^E \bmod M$. So, the same operation, i.e., modular exponentiation, is used to perform both processes: encryption and decryption. The operation consists of a repetition of modular multiplications. The subject of hardware

[*] Corresponding author. Tel.: +55 21 2587 7442; fax: +55 21 2587 7374.
E-mail addresses: nadia@eng.uerj.br (N. Nedjah), ldmm@eng.uerj.br (L. de Macedo Mourelle).
[1] The authors are grateful to FAPERJ and CNPq for their financial support.

implementation of the RSA cryptosystem is widely studied [3,4].

The performance of public-key cryptosystems is primarily determined by the implementation efficiency of the modular multiplication and exponentiation. As the operands (the plain text or cipher/possibly partially ciphered text) are usually large. So, in order to improve time requirements of the encryption and decryption operations, it is essential to attempt to minimise the number of modular multiplications performed and reduce the time requirements of a single modular multiplication [4–6].

There are various algorithms that implement modular multiplication such as Barrett–Booth's method [7,8] and Brickell's algorithm [3,9]. Here, we always use Montgomery's algorithm [5] as it is considered the most efficient [10]. The attractive feature of this method is that it computes modular multiplications without trial divisions [5]. Essential improvements of Montgomery's algorithm are presented in [11], which was further improved in [12] by reducing the number of necessary iterations.

In general terms, the m-ary method for exponentiation [2] may be thought of as a three-step procedure: (i) partitioning the binary representation of exponent E in ℓ-bit windows; (ii) pre-computing necessary powers; (iii) iterating the squarings of the partial result ℓ times to shift it over, and then multiplying it by the power indicated in the next window, if it is different from 0. The details of the m-ary method are given in Algorithm 1, whereby $C = T^E \bmod M$ is computed.

Algorithm 1. m-ary(T,M,E)

```
1: pre-compute and store $T^k$ for all $k = 2, 3,4,\ldots, 2^\ell - 1$;
2: partition $E$ into $p$ windows of $\ell$ bits;
3: $C := T^{V_{p-1}} \bmod M$;
4: for $i := p - 2$ downto 0 do
5:    $C := C^{2^\ell} \bmod M$;
6:    if $V_i \neq 0$ then $C := C \times T^{V_i} \bmod M$;
7: return $C$;
```

end.

Note that the algorithm does not give any details on how the pre-processing step (line 1 of Algorithm 1) should be done. One can compute all non-trivial $2^\ell - 2$ possible powers (excluding 0 and 1), which requires $2^\ell - 2$ modular multiplications. However, one can also try to optimise this step by computing only the necessary powers, i.e., those that actually appear in the exponent, which should reduce the number of modular multiplications. Algorithm 2 mirrors this modification.

For instance, let exponent $E = 343$ be partitioned as follows: $E = \underbrace{0001}\underbrace{0101}\underbrace{0111}$. The 16-ary exponentiation method would require 14 modular multiplications in the pre-processing step to compute all powers (T^2, $T^3,\ldots,T^{15}$). Then it would require 8 modular multiplications in the squaring step (line 5 in Algorithm 1) and 2 others in the multiplying step (line 6 of Algorithm 1), which amounts to a total of 24 modular multiplications. For the same exponent, the adaptive 16-ary exponentiation method would require only 4 modular multiplications in the pre-processing step to compute powers T^2, T^3, T^5 and T^7. The exponentiation step being the same, the number of modular multiplications would then amount to a total of 14 operations. Here, both of the strategies described in Algorithms 1 and 2 are implemented and the designs analysed in terms of area and time requirements.

Algorithm 2. Adaptive m-ary(T,M,E)

```
1: partition $E$ into $p$ windows of $\ell$ bits;
2: pre-compute and store $T^{V_0}$, $T^{V_1}$, ..., $T^{V_{p-1}}$;
3: $C := T^{V_{p-1}} \bmod M$;
4: for $i := p$-2 downto 0 do
5:    $C := C^{2^\ell} \bmod M$;
6:    if $V_i \neq 0$ then $C := C \times T^{V_i} \bmod M$;
7: return $C$;
```

end.

The rest of this paper is organised as follows: in Section 2, we describe Montgomery's algorithm, which is used to implement the modular multiplication, as well as the architecture of the designed hardware; in the subsequent two sections, i.e., Sections 3 and 4, we discuss the architecture and implementation of each of the proposed designs for modular exponentiation as explained earlier; thereafter, in Section 5, we present some performance figures in order to assess the efficiency of the proposed implementations; finally, in Section 6, we summarise the work presented throughout the paper, draw some useful conclusions and present some directions for future work.

2. Montgomery's modular multiplication

Algorithms that formalise the operation of modular multiplication generally consist of two steps:

the first generates the product $P = A \times B$ and the second reduces this product P *modulo* M.

The straightforward way to implement a multiplication is based on an iterative adder-accumulator for generating partial products. However, this solution is quite slow as the final result is only available after n clock cycles, where n is the size of the operands [13].

A faster version of the iterative multiplier should add several partial products at once. This could be achieved by unfolding the iterative multiplier and yielding a combinatorial circuit that consists of several partial product generators together with several adders that operate in parallel [13].

One of the widely used algorithms for efficient modular multiplication is Montgomery's algorithm [3,10,14–16]. This algorithm computes the product of two integers *modulo* a third one without performing any trial divisions. It yields the reduced product using a series of additions.

Let A, B and M be the multiplicand, the multiplier and the modulus, respectively, and let n be the number of bits in their binary representations. So, we denote A, B and M as follows:

$$A = \sum_{i=0}^{n-1} a_i \times 2^i, \quad B = \sum_{i=0}^{n-1} b_i \times 2^i,$$
$$M = \sum_{i=0}^{n-1} m_i \times 2^i, \qquad (1)$$

The pre-conditions for the application of Montgomery's algorithm are as follows: (i) the modulus M needs to be relatively prime to the radix, i.e., there exists no common divisor for M and the radix; (ii) the multiplicand and the multiplier need to be smaller than M. As we use the binary representation of the operands, then the modulus M needs to be odd to satisfy the first pre-condition. Montgomery's modular multiplication is described in Algorithm 3.

Algorithm 3. Montgomery(A,B,M)

```
1: $R$ :=0;
2: **for** $i$ :=0 to $n-1$ **do**
3:     $R := R + a_i \times B$;
4:     **if** $r_0 = 0$ **then** $R := R$ div 2
6:     **else** $R := (R + M)$ div 2;
7: **return** $R$;
```

end.

The detailed architecture of the Montgomery modular multiplier is given in Fig. 1. It uses two multiplexers, two adders, two shift registers, three registers and a controller. This modular multiplier is used in all the subsequently described implementations of modular exponentiation presented in this paper.

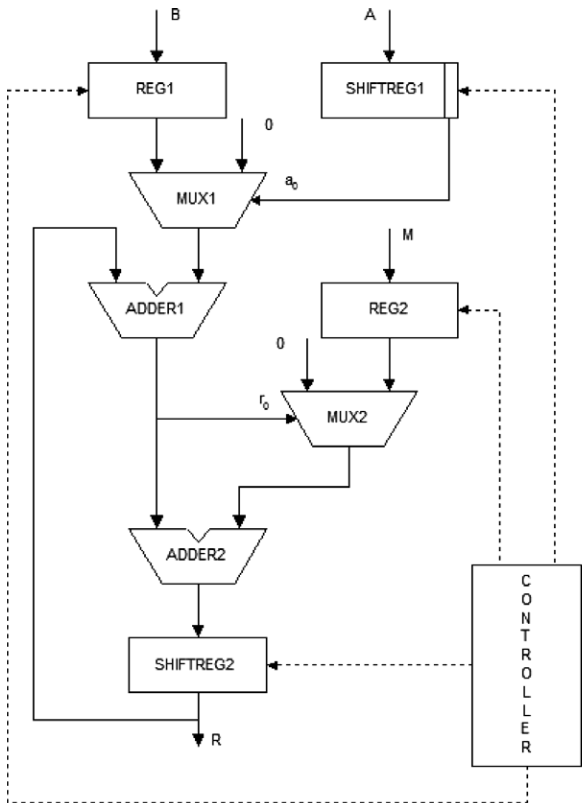

Fig. 1. Architecture of Montgomery's multiplier.

Multiplexer MUX1 of 1 passes through 0 or B depending on whether the less significant bit a_0 of operand A indicates 0 or 1, respectively. Multiplexer MUX2 passes 0 or M depending on whether bit r_0, which is the less significant bit of partial result R (line 3 of Algorithm 3), indicates 0 or 1, respectively. Adder ADDER1 delivers the sum $R + a_i \times B$ and adder ADDER2 yields the sum $R + M$ (line 6 of the same algorithm). Shift register SHIFTREG1 provides bit a_i. In each iteration i of the multiplier, this shift register is right-shifted once so that a_0 contains a_i. Shift register SHIFTREG2 implements the division by 2 (line 6 of Algorithm 3).

The role of the controller consists of synchronizing the shifting and loading operations of the registers. It also controls the number of iterations that have to be performed. For this end, the controller uses a simple down-counter that starts from n. The counter is inherent to the controller. In order

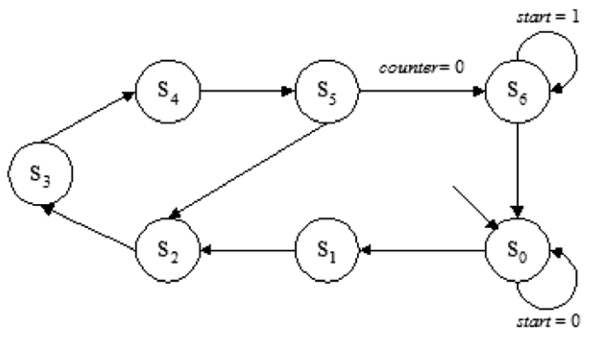

Fig. 2. The state transtion digram of the controller used by the modular multiplier.

to synchronize the work of the architecture components, the controller consists of a state machine, which has 7 states defined as follows. The state transition diagram of the controller is shown in Fig. 2.

- S_0: initialise the system; if $start = 1$ then go to S_1;
- S_1: load B into REG1; load M into REG2; load A into SHIFTREG1; reset SHIFTREG2;
- S_2: wait for ADDER1; wait for ADDER2; decrement *counter*;
- S_3: load partial result into SHIFTREG2; decrement counter;
- S_4: right-shift SHIFTREG1; right-shift SHIFTREG2;
- S_5: if $counter = 0$ then go to S_6 else go to S_2;
- S_6: indicate end of operation; if $start = 0$ then go to S_0;

3. Hardware implementation of the *M*-ary method

The hardware that implements the *m*-ary method, as described in Algorithm 1, is given in Fig. 3. The pre-processing step computes all the possible powers of T, with respect to the partition size ℓ, and stores them in a local memory (MEM). Later on, i.e., in the exponentiation step, each partition of the exponent E is used to address the memory, read and use the pre-computed power, as defined in line 3 of Algorithm 1.

There is no need to store T^0 mod M, since zero partitions are not considered as they do not change the partial result (see line 6 of Algorithm 1). The first power of T, i.e., T^2 mod M, is computed by passing T through both multiplexers MUX1 and MUX4, feeding the Modular Multiplier of Fig. 1. The result is then stored in location 2 of MEM, using the initial value of register REGI. This register is responsible for generating the power memory addresses during the pre-processing step. The subsequent possible powers are obtained successively, by passing the previous result through multiplexers MUX3 then MUX1. Plaintext T is kept available through multiplexer MUX4. The memory locations are generated by incrementing REGI whenever a new address is required.

In each iteration of the exponentiation step, the partial result C is raised to the power 2^ℓ then multiplied by T^{V_i} mod M, whenever V_i is not a zero partition (see lines 6 of Algorithm 1). The values of T^{V_i} mod M are obtained from the power memory, according to the current partition in exponent E. In order to obtain the value of the current partition, we store exponent E in a shift register, namely REGE, from which the most significant partition is retrieved to address the power memory (see line 3 and 6 of Algorithm 1). When a new partition is required, register REGE is left-shifted ℓ times. Recall that ℓ represents the partition size. This operation is controlled by a down-counter REGL, which com-

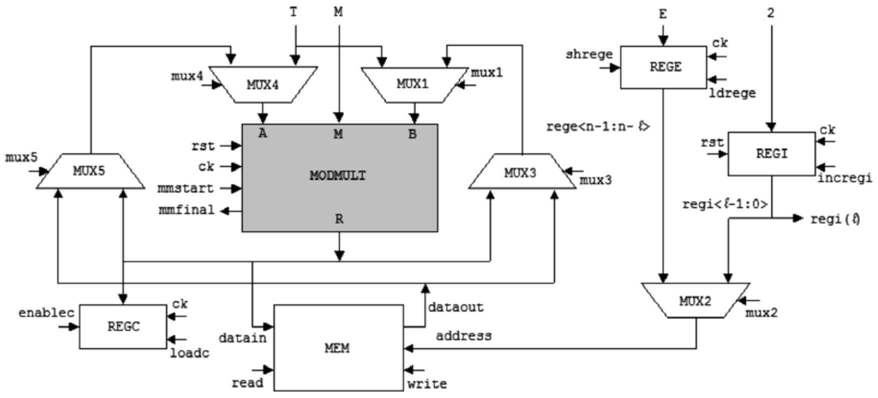

Fig. 3. Hardware architecture of the *m*-ary method as described in Algorithm 1.

mences with ℓ as initial state and is decremented each time the register REGE is left-shifted.

The square-and-multiply loop (starting in line 4 of Algorithm 1) consists of two main phases: (i) the first phase performs ℓ squaring operations of the partial result. For this purpose, the partial result is fed-back to inputs A and B of the modular multiplier of Fig. 1, through multiplexers MUX3 and MUX5, and then MUX1 and MUX4, respectively. The squaring operation is also controlled by the down-counter REGL, which starts with ℓ as initial state and is decremented each time a single squaring operation is completed. Note that a new partition is acquired before the squaring operation occurs so the counter REGL can be used to control both operations; (ii) the second phase performs the modular multiplication of the partial result with a pre-computed power, when the current partition is not zero. The power of T, i.e., T^{V_i} mod M, is read from the power memory, at the location indicated by the most significant partition of register REGE. The address is passed through multiplexer MUX2.

The square-and-multiply loop is performed until the least significant partition of E is reached. This is also controlled by another down-counter REGP, which commences with p and is decremented each step of the loop. Recall that p denotes the number of partitions in exponent E. Whenever the down-counter reaches zero, the final result is subsequently loaded from SHIFTREG2 (see Fig. 1) into register REGC. Apart from the clock (ck) and reset (rst), all the other signals used in the architecture of Fig. 3 are provided by a controller. Now, we concentrate on its description.

The pre-processing step consists of performing $2^\ell - 2$ modular multiplications, while the exponentiation step consists of computing $\ell(p-1)+q$, $(0 \leqslant q \leqslant p-1)$ modular multiplications, wherein q is the number of non-zero partitions. The operands, however, differ from one multiplication to another. The main work of the controller consists of setting up the right operands for each one of these modular multiplications. When the operands are ready, the controller asserts signal *mmstart*, which activates the modular multiplier. The controller then waits for the completion of the operation, which is indicated by the assertion of signal *mmfinal* by the multiplier. Consequently, the controller withdraws signal *mmstat* and so does the multiplier with signal *mmfinal* thus completing the handshake protocol. The description of each state of the controller is presented as follows, wherein MSP stands for the most significant partition. Counters REGP and REGL are not depicted in the architecture of Fig. 3 as these are embedded in the controller. The state transition digaram of the controller is shown in Fig. 4.

- S_0: initialise the system; if $start = 1$ then go to S_1;
- S_1: load REGE with E;
  load REGL with ℓ;
  load REGP with p; load REGI with 2;
- S_2: start modular multiplier;
- S_3: if the modular multiplier has finished then go to S_4;
- S_4: stop modular multiplier;
  write the result into MEM(REGI);
- S_5: increment REGI;
- S_6: if REGI(ℓ) = 1 then go to S_8 else go to S_7;
- S_7: start modular multiplier; goto S_3;
- S_8: read MEM(MSP of REGE); decrement REGP;
- S_9: start modular multiplier;
- S_{10}: if modular multiplier finished go to S_{11};
- S_{11}: stop modular multiplier; decrement REGL;
- S_{12}: if REGL = 0 then go to S_{13} else go to S_9;
- S_{13}: load REGL with ℓ;
- S_{14}: decrement REGL; left shift REGE;
- S_{15}: if $\text{REG}_\ell = 0$ then
  if MSP of REGE $\neq 0$ then go to S_{16}
  else if REGP $\neq 0$ then go to S_9 else go to S_{20};
- S_{16}: read MEM(MSP of REGE);
- S_{17}: start modular multiplier;
- S_{18}: if modular multiplier finished then
  if REGP = 0 then go to S_{20} else go to S_{19};
- S_{19}: decrement REGP; go to S_9;
- S_{20}: indicate end of operation;
  load REGC with final result;
  if $start = 0$ then go to S_0;

The computation of T^2 mod M requires T to be sent to both inputs A and B of the modular multiplier (see Fig. 1). This means that multiplexers MUX1 and MUX4 are prepared to do so, having their control signals mux_1 and mux_4 set accordingly, during state S_2. For the following power computations, we pass the partial result through MUX1, instead of T. This changes the control signal mux_1 and, therefore, we can not reuse state S_2. Note that REGI consists of $\ell + 1$ bits instead of ℓ. The most significant extra bit goes to 1 when all possible powers have been considered. So, in state S_6, the controller checks the completion of the pre-processing step by simply reading bit ℓ in REGI. In state S_{15}, the controller checks whether the partition is not zero by

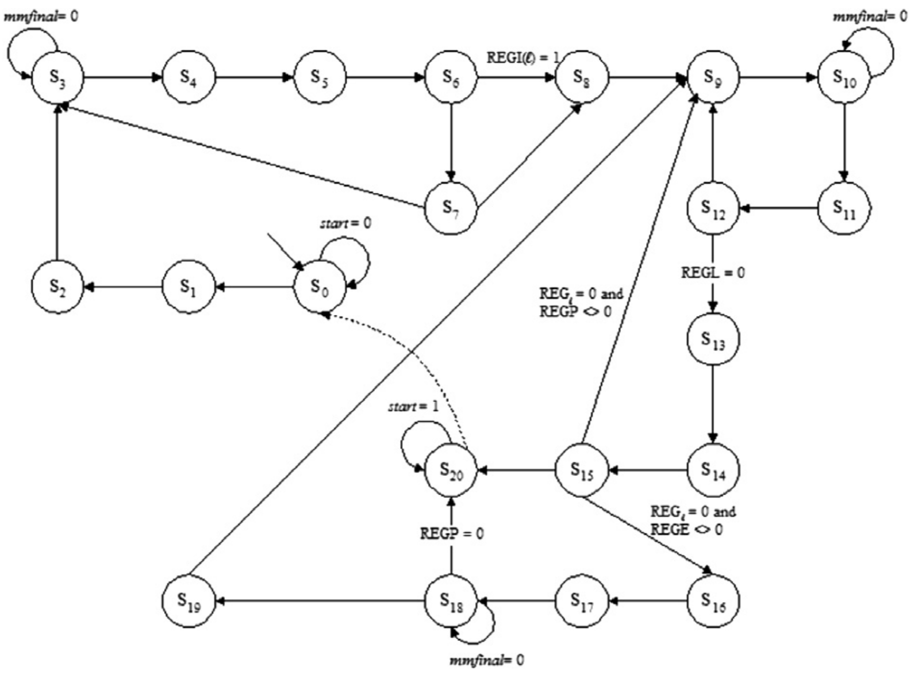

Fig. 4. The state transtion digram of the controller used by the first hardware implementation.

reading a control signal that is the ORed signal of the bits of MSP.

4. Implementation of the adaptive *M*-ary method

The architecture for the adaptive *m*-ary hardware that computes the necessary powers in the exponentiation step only is given in Fig. 5. The pre-processing step computes the required powers, based on the addition chain [17] provided *a priori*, and stores these powers in a local memory (MEM). The addition chain prescribes the sequence in which the powers must be computed. The power to compute next is always the product of two previously computed ones. Each position of memory MEM consists of two data: the ℓ high-order bits store the exponent, as provided by the addition sequence, and the remaining *n* low-order bits store the corresponding power. At the beginning, position 0 contains value 1 in its high-order bits and *T* in its low-order bits.

The powers are computed according to the addition chain rule: $a_k = a_i + a_j$, $0 \leqslant i \leqslant j < k$. Indexes *i*, *j* and *k* are stored in registers REGADDRK, REGADDRI and REGADDRJ, which start with initial value 2, 1 and 1, respectively. The memory locations addressed by REGADDRK is read and the high-order bits of the word are loaded into register REGK. The memory location addressed by REGADDRI and REGADDRJ are, subsequently, read and the word loaded into REGI (high-order bits) and REGA (low-order bits), and in REGJ (high-order bits) and REGB (low-order bits), respectively. The sum of REGI and REGJ is then compared to the content of REGK. Whenever the comparison is valid and so correct REGADDRI and REGADDRJ are found, the current contents of registers REGA and REGB are thus modular multiplied and the result stored in memory location REGADDRK. Subsequently, register REGADDRK is incremented and, if there are elements in the addition chain yet to be considered, a new value for register REGK is obtained from the memory MEM. Registers REGADDRI and REGADDRJ are loaded with the content of REGADDRK. The search for other valid values begins, by successively decrementing REGADDRI and REGADDRJ, until the right *i* and *j* are encountered. Note that such indexes are always found as an appropriate addition chain is used.

Each partition of the exponent *E* is used to address the memory to obtain the corresponding pre-computed power of *T* (mod *M*). Memory MEM is in fact associative. This allows us to read the data based on the value of the current partition. Signal *match* is asserted for every read cycle of MEM.

The pre-processing step performs only a subset of the modular multiplications computed by that design. The square-and-multiply step is performed

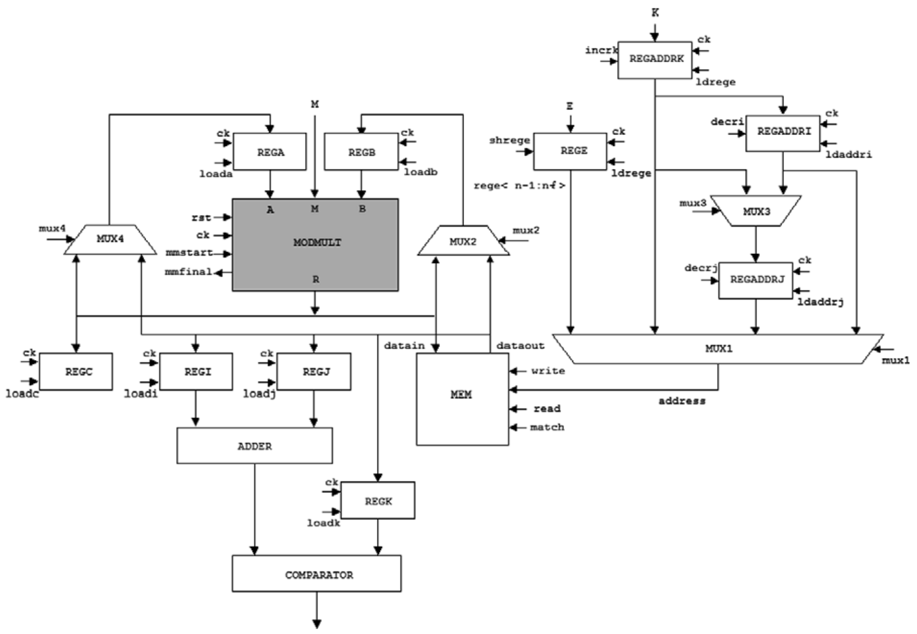

Fig. 5. Hardware architecture of the m-ary method as described in Algorithm 2.

in the same way it is done in the previous design (see Fig. 3). As before, the hardware needs a controller, whose role consists of setting up the right operands for each one of the performed modular multiplications. It has 9 more states compared with the controller of the hardware described in the previous section. In the state machine described in the following and as before, MSP stands for the most significant partition. Register REGP, REGL and REGS are not depicted in the architecture of Fig. 5 as these are embedded in the controller. The state transition digaram of the controller is shown in Fig. 6.

- S_0: initialise the system;
  if $start = 1$ then go to S_1;
- S_1: load REGE with E; load REGADDRK with 2;
  load REGP with p; load REGL with ℓ;
  load REGS with s;

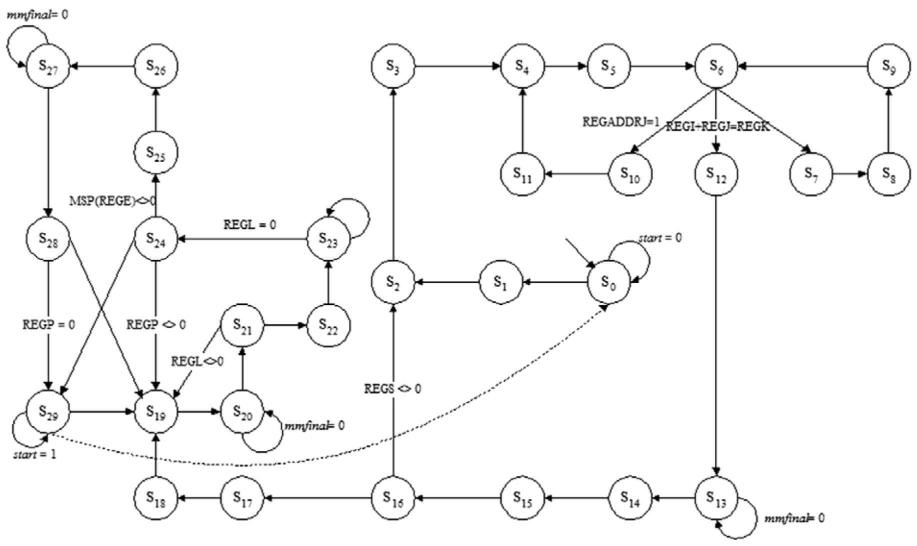

Fig. 6. The state transtion digram of the controller used by the second hardware implementation.

- S_2: read MEM(REGADDRK);
  load REGADDRI with REDADDRK;
  load REGADDRJ with REGADDRK;
  decrement REGS;
- S_3: load REGK with high-order bits of MEM(REGADDRK);
  decrement REGADDRI; decrement REGADDRJ;
- S_4: read MEM(REGADDRI);
- S_5: load REGI with high-order bits of MEM(REGADDRI);
  load REGJ with high-order bits of MEM(REGADDRI);
  load REGA with low-order bits of MEM(REGADDRI);
  load REGB with low-order bits of MEM(REGADDRI);
- S_6: if (REGI + REGJ) = REGK then go to S_{12};
  else if REGADDRJ = 1 then go to S_{10} else go to S_7;
- S_7: decrement REGADDRJ;
- S_8: read MEM(REGADDRJ);
- S_9: load REGJ with high-order bits of MEM(REGADDRJ);
  load REGB with low-order bits of MEM(REGADDRJ);
  go to S_6;
- S_{10}: decrement REGADDRI;
- S_{11}: load REGADDRJ with REGADDRI; go to S_4;
- S_{12}: start the modular multiplier;
- S_{13}: if modular multiplier has finished then go to S_{14};
- S_{14}: stop the modular multiplier;
- S_{15}: write the result in MEM(REGADDRK);
- S_{16}: increment REGADDRK; decrement REGS;
  if REGS $\neq$ 0 then go to S_2 else go to S_{17};
- S_{17}: read MEM(MSP of REGE); decrement REGP;
- S_{18}: load REGA with low-order bits of MEM(MSP of REGE);
  load REGB with low-order bits of MEM(MSP of REGE);
- S_{19}: start modular multiplier;
- S_{20}: if modular multiplication finished then go to S_{21};
- S_{21}: stop modular multiplier; load REGA with R;
  load REGB with R; decrement REGL;
  if REGL $\neq$ 0 then got to S_{19} else go to S_{22}
- S_{22}: decrement REGP; load REGL with ℓ;

Table 1
Area and time requirements for the first two implementations

n	ℓ	1st Implementation		2nd Implementation	
		Area	Time	Area	Time
64	2	345	18.1	501	3.7
	3	392	16.1	591	3.1
	4	404	15.4	804	2.9
128	2	767	23.5	910	2.9
	3	811	22.0	1047	6.1
	4	821	20.1	1223	5.8

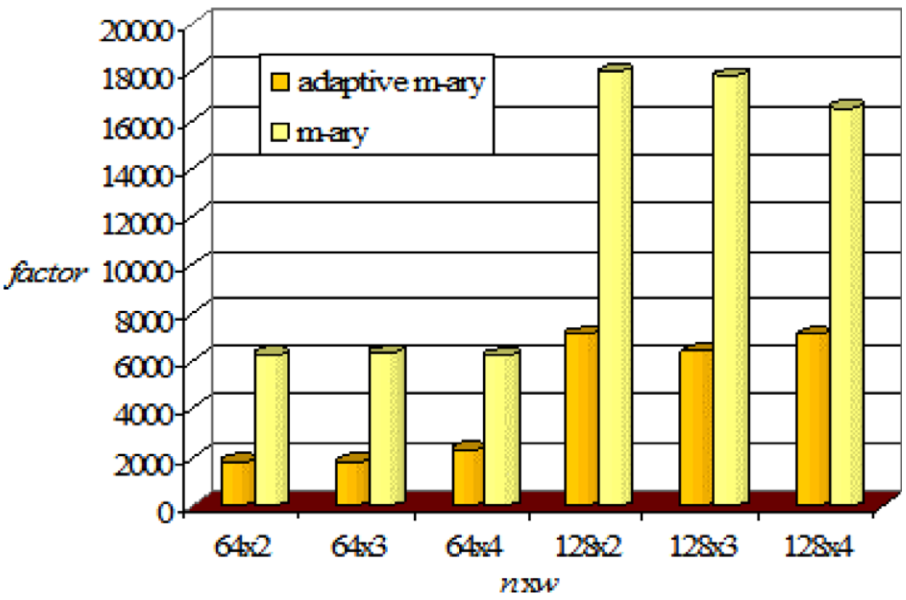

Fig. 7. Trade-off for the first two implementations.

- S_{23}: decrement REGL; left shift REGE;
  if REGL $= 0$ then go to S_{24};
- S_{24}: load REGL with ℓ;
  if MSP of REGE $\neq 0$ then go to S_{25}
  else if REGP $\neq 0$ then go to S_{19} else go to S_{29};
- S_{25}: load REGB with low-order bits of
  MEM(MSP of REGE);
- S_{26}: start modular multiplier;
- S_{27}: if modular multiplier has finished then go to S_{28}
- S_{28}: stop modular multiplier; load REGA with R;
  load REGB with R;
  if REGP $\neq 0$ then go to S_{19} else go to S_{29};
- S_{29}: load REGC with R; indicate end of operation;
  if *start* $= 0$ then Go to S_0;

5. Performance results

The implementations were implemented using reconfigurable FPGAs of the VIRTEX family, which includes a PowerPC processor [18]. The area and time requirements are reported in Table 1 for the first two implementations, wherein ℓ denotes the partition size. In Table 1, the performance factor $area \times time$ shows that the trade-off is much better for the implementation of the adaptive m-ary method, which pre-computes only the necessary modular powers than for that of the m-ary method, which pre-computes all possible ones. This can be clearly seen in the chart of Fig. 7. Note that the requirements in terms time in the second implementation were reduced but those in terms of hardware area are slightly larger than that for the first implementation.

6. Conclusions

In this paper, we presented four alternative implementations for the modular exponentiation based on the m-ary and addition chain-based methods. For all the proposed designs, we used the same implementation for the modular multiplication. The modular multiplier is an iterative efficient implementation of Montgomery's algorithm.

The first proposed design for the modular exponentiation is inspired by the m-ary method. The implementation is compact but computes more powers than necessary and therefore consumes a considerable amount of time to complete the operation. The second design is similar to the first one as it is also inspired by the m-ary method. The exponentiation phase is the same. However, the pre-processing based on the exponent is optimised as only required modular powers are pre-computed. The pre-processing is done using an appropriate addition chain. For doing so, its implementation is fast but consumes slightly more area than the first implementation. Nevertheless, the trade-off between hardware area and response time indicates the second implementation as a better solution.

This study reveals that the choice of the more adequate implementation for modular exponentiation depends closely on the available resources and the targeted aims: area vs. encryption/decryption throughput. In future work, we intend to investigate the impact of the operand size and partition size on the area and time requirements of the proposed designs.

Acknowledgments

We are grateful to the reviewers that contributed to the great improvement of the original version of this paper with their valuable comments and suggestions. We also are thankful to FAPERJ (Fundação de Amparo à Pesquisa do Estado do Rio de janeiro) and CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) for their continuous financial support.

References

[1] R. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signature and public-key cryptosystems, Communications of the ACM 21 (1978) 120–126.

[2] Ç.K. Koç, High-speed RSA Implementation, Technical report, RSA Laboratories, Redwood City, CA, USA, November 1994.

[3] E.F. Brickell, A survey of hardware implementation of RSA, in: G. Brassard (Ed.), Advances in Cryptology, Proceedings of CRYPTO'98, Lecture Notes in Computer Science, vol. 435, Springer-Verlag, 1989, pp. 368–370.

[4] C.D. Walter, Systolic modular multiplication, IEEE Transactions on Computers 42 (3) (1993) 376–378.

[5] P.L. Montgomery, Modular multiplication without trial division, Mathematics of Computation 44 (1985) 519–521.

[6] N. Nedjah, L.M. Mourelle, Two hardware implementations for the Montgomery multiplication: sequential vs. parallelProc. of the 15th. Symposium on Integrated Circuits and Systems Design, IEEE Computer Society Press, 2002, pp. 3–8.

[7] A. Booth, A signed binary multiplication technique, Quarterly Journal of Mechanics and Applied Mathematics (1951) 236–240.

[8] G.W. Bewick, Fast multiplication algorithms and implementation, Ph.D. Thesis, Department of Electrical Engineering, Stanford University, USA, 1994.

[9] C.D. Walter, A verification of Brickell's fast modular multiplication algorithm, International Journal of Computer Mathematics 33 (1990) 153–169.

[10] S.E. Eldridge, C.D. Walter, Hardware implementation of Montgomery's Modular Multiplication Algorithm, IEEE Transactions on Computers 42 (6) (1993) 619–624.

[11] C.D. Walter, Montgomery exponentiations needs no final subtractions, Electronics Letter 35 (21) (1999) 1831–1832.

[12] G. Hachez, J.J. Quisquater, Montgomery exponentiations needs no final subtractions: improved results, Cryptographic Hardware and Embedded Systems, in: C. Paar, Ç.K. Koç (Eds.), Proceedings of CHES'00, Lecture Notes in Computer Science, vol. 1965, Springer-Verlag, 2000, pp. 293–301.

[13] J. Rabaey, Digital integrated circuits: A design perspective, Prentice-Hall, 1995.

[14] N. Nedjah, L.M. Mourelle, Yet another implementation of modular multiplication, in: Proceedings of 13th Symposium of Computer Architecture and High Performance Computing, 2001, pp. 70–75.

[15] N. Nedjah, L.M. Mourelle, Reduced hardware architecture for the Montgomery modular multiplication, in: Proceedings of the third IEEE Symposium Mathematical Methods and Computational Techniques in Electrical Engineering, Athens, Greece, 2001.

[16] N. Nedjah, L.M. Mourelle, Reconfigurable hardware implementation of Montgomery modular multiplication and parallel binary exponentiationProceedings of the Euromicro Symposium on Digital System Design, Dortmund, Germany, IEEE Computer Society Press, 2002, pp. 226–235.

[17] N. Nedjah, L.M. Mourelle, Efficient Pre-processing for Large Window-based Modular Exponentiation using Genetic Algorithms, Lecture Notes of Computer Science vol. 2718 (2003) 625–635.

[18] Xilinx™ Inc. Foundation Series, http://www.xilinx.com.

Nadia Nedjah is an associate professor in the Department of Electronics Engineering and Telecommunications at the Faculty of Engineering, State University of Rio de Janeiro, Brazil. Her research interests include functional programming, embedded systems and reconfigurable hardware design as well as cryptography. Nedjah received her PhD in Computation from the University of Manchester – Institute of Science and Technology (UMIST), England, her MSc in System Engineering and Computation from the University of Annaba, Algeria and her Engineerind degree in Computer Science also from the University of Annaba, Algeria.

Luiza de Macedo Mourelle is an associate professor in the Department of System Engineering and Computation at the Faculty of Engineering, State University of Rio de Janeiro, Brazil. Her research interests include computer architecture, embedded systems design, hardware/software codesign and reconfigurable hardware. Mourelle received her PhD in Computation from the University of Manchester – Institute of Science and Technology (UMIST), England, her MSc in System Engineering and Computation from the Federal University of Rio de Janeiro (UFRJ), Brazil and her Engineering degree in Electronics also from UFRJ, Brazil.