Efficient Hardware Implementation of Modular Multiplication and Exponentiation for Public-Key Cryptography

Nadia Nedjah and Luiza de Macedo Mourelle

Department of Systems Engineering and Computation
Faculty of Engineering, State University of Rio de Janeiro
(nadia,ldmm)@eng.uerj.br

Abstract. Modular multiplication and modular exponentiation are fundamental operations in most public-key cryptosystems such as RSA and DSS. In this paper, we propose a novel implementation of these operations using systolic arrays based architectures. For this purpose, we use the Montgomery algorithm to compute the modular product and the left-to-right binary exponentiation method to yield the modular power. Our implementation improves time requirement as well as the time×area factor when compared that of Blum's and Paar's.

1 Introduction

RSA [1] is a strong public-key cryptosystem, which is generally used for authentication protocols. It consists of a set of three items: a *modulus* M of around 1024 bits and two integers D and E called *private* and *public* keys that satisfy the property $T^{DE} \equiv T \bmod M$. Plain text T obeying $0 \leq T < M$. Messages are encrypted using the public key as $C = T^E \bmod M$ and uniquely decrypted as $T = C^D \bmod M$. So the same operation is used to perform both processes: encryption and decryption. The modulus M is chosen to be the product of two large prime numbers, say P and Q. The public key E is generally small and contains only few bits set (i.e. bits = 1), so that the encryption step is relatively fast. The private key D has as many bits as the modulus M and is chosen so that $DE = 1 \bmod (P-1)(Q-1)$. The system is secure as it is computationally hard to discover P and Q. It has been proved that it is impossible to break an RSA cryptosystem with a modulus of 1024-bit or more.

The modular exponentiation is a common operation for scrambling and is used by several public-key cryptosystems, such as the RSA encryption scheme [1]. It consists of a repetition of modular multiplications: $C = T^E \bmod M$, where T is the plain text such that $0 \leq T < M$ and C is the cipher text or vice-versa, E is either the public or the private key depending on whether T is the plain or the cipher text. The decryption and encryption operations are performed using the same procedure, i.e. using the modular exponentiation.

The performance of such cryptosystems is primarily determined by the implementation efficiency of the modular multiplication and exponentiation. As the

J.M.L.M. Palma et al. (Eds.): VECPAR 2002, LNCS 2565, pp. 451-463, 2003.

operands (the plain text of a message or the cipher or possibly a partially ciphered) text are usually large (i.e. 1024 bits or more), and in order to improve time requirements of the encryption/decryption operations, it is essential to attempt to minimise the number of modular multiplications performed and to reduce the time requirement of a single modular multiplication. Hardware implementations of RSA cryptosystems are widely studied as in [2, 3, 4, 5, 6, 7, 8].

Most proposed implementations of RSA system are based on Montgomery modular algorithm. There have been various proposals for systolic array architectures for modular multiplication [3, 5, 6, 7] and for modular exponentiation [5, 7]. But as far as we know, only one physical implementation has been reported including numerical figures of space and time requirements for different sizes of the exponent. It is that of Blum and Paar [6]. Tiountchik and Trichina [7] provided only one numerical figure, which is for a 132-bit modulus cryptosystem. It needs 4 Kgates FPGA chip.

In contrast with Tiountchik's and Trichina's and Walter's systolic version for modular multiplication which uses two full-adders per processing element, in our our implementation, a processing element uses at most one full adder. Some processing elements use only a half adder or a simple and-gate as it will be shown in details in Section 3. So, in general, our processing element has a simpler architecture and thus saves hardware space as well as response time.

All hardware implementations of modular exponentiation reported so far as well as the present one use the binary method as it constitutes the unique method suitable for hardware implementation. Other exponentiation methods exist (see Section 4).

Unlike Tiountchik's and Trichina's systolic version for modular exponentiation, our implementation uses more hardware space but avoids extra inputs and analysis to decide what to do: multiplication or squaring. In contrast with Blum's and Paar's implementation of exponentiation is that theirs is iterative and thus consumes more time but less space area.

In the rest of this paper, we start off by describing the algorithms used to implement the modular operation. Then we present the systolic architecture of the modular multiplier. Then we describe the systolic architecture for the exponentiator, which based on the binary exponentiation method. Finally, we compare the space and time requirements of our implementation, produced by the Xilinx project manager with those produced for Blum's and Paar's implementation.

2 Montgomery Algorithm

Algorithms that formalise the operation of modular multiplication generally consist of two steps: one generates the product $P = A \times B$ and the other reduces this product modulo M, i.e. $R = P \bmod M$.

The straightforward way to implement a multiplication is based on an iterative adder-accumulator for the generated partial products. However, this solution is quite slow as the final result is only available after n clock cycles, n is the size of the operands [7].

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 [8] and [9].

One of the widely used algorithms for efficient modular multiplication is the Montgomery's algorithm [10, 11] and [12]. This algorithm computes the product of two integers modulo a third one without performing division by M and it is described in Figure 1. It yields the reduced product using a series of additions

Let A, B and M be the multiplicand and multiplier and the modulus respectively and let n be the number of digit in their binary representation, i.e. the *radix* is 2. So, we denote A, B and M as follows:

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

The pre-conditions of the Montgomery algorithm are as follows:

- The modulus M needs to be relatively prime to the *radix*, i.e. there exists no common divisor for M and the radix; (As the radix is 2, then M must be an odd number.)
- 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.

The Montgomery algorithm uses the least significant digit of the accumulating *modular partial product* to determine the multiple of M to subtract. The usual multiplication order is reversed by choosing multiplier digits from least to most significant and shifting down. If R is the current modular partial product, then q is chosen so that $R+q\times M$ is a multiple of the radix r, and this is right-shifted by one position, i.e. divided by r for use in the next iteration. Consequently, after n iterations, the result obtained is $R = A\times B\times r^{-n} \bmod M$. A version of Montgomery algorithm is given in Figure 1 where r_0 is the less significant bit of the partial modular product or residue R.

In order to yield the exact result, we need an extra Montgomery modular multiplication by the constant $r^n \bmod M$. As we use binary representation of numbers, we need to compute $(2^n \bmod M)\times R \bmod M$. We look at in the next section.

```
algorithm Montgomery(A, B, M)
        int R ← 0;
        1: for i = 0 to n-1
        2:      R ← R + a_i×B;
        3:      if R mod 2 = 0 then
        4:              R ← R div 2
        5:      else
        6:              R ← (R + M) div 2;
return R;
```

Fig. 1. Montgomery modular algorithm

3 Systolic Montgomery Modular Multiplication

A modified version of Montgomery algorithm is that of Figure 2. The least significant bit of $R + a_i \times B$ is the least significant bit of the sum of the least significant bits of R and B if a_i is 1 and the least significant bit of R otherwise. Furthermore, new values of R are either the old ones summed up with $a_i \times B$ or with $a_i \times B + q_i \times M$ depending on whether q_i is 0 or 1.

Consider the expression $R + a_i \times B + q_i \times M$ of line 2 in the algorithm of Figure 2. It can be computed as indicated in the last column of the Table 1 depending on the value of the bits a_i and q_i.

A bit-wise version of the algorithm of Figure 2, which is at the basis of our systolic implementation, is described in Figure 3.

Table 1. Computation of $R + a_i \times B + q_i \times M$

a_i	q_i	$R + a_i \times B + q_i \times M$
1	1	$R + MB$
1	0	$R + B$
0	1	$R + M$
0	0	R

```
algorithm ModifiedMontgomery(A, B, M) {
        int R ← 0;
        1: for i = 0 to n-1
        2:      q_i  ←  (r_0 + a_i×b_0) mod 2;
        3:      R    ←  (R + a_i×B + q_i×M) div 2;
return R;
```

Fig. 2. Modified Montgomery modular algorithm

```
algorithm SystolicMontgomery(A, B, M, MB)
        int  R ← 0; bit  carry ← 0, x;
        0: for i = 0  to  n
        1:      q_i  ←  r_0^(i) ⊕ a_i.b_0;
        2:      for j = 0 to n
        3:           switch a_i, q_i
        4:                1,1:  x ← mb_i;
        5:                1,0:  x ← b_i;
        6:                0,1:  x ← m_i;
        7:                0,0:  x ← 0;
        8:           r_j^(i+1)  ← r_(j+1)^(i) ⊕ x_i ⊕ carry;
        9:           carry      ← r_(j+1)^(i) .x_i + r_(j+1)^(i) .carry + x_i.carry;
return R;
```

Fig. 3. Systolic Montgomery modular algorithm

All algorithms, i.e. those of Figure 1, Figure 2 and Figure 3 are equivalent. They yield the same result. In the algorithm above *MB* represents the result of $M + B$, which has at most has n+1 bits.

Assuming the algorithm of Figure 3 as basis, the main processing element (PE) of the systolic architecture of the Montgomery modular multiplier computes a bit r_j of residue R. This represents the computation of line 8. The left border PEs of the systolic arrays perform the same computation but beside that, they have to compute bit q_i as well. This is related to the computation of line 1. The duplication of the PEs in a systolic form implements the iteration of line 0. The systolic architecture of the modular Montgomery multiplier is shown in Figure 4.

The architecture of the basic PE, i.e. $cell_{i,j}$ $1 \le i \le n-1$ and $1 \le i \le n-1$, is shown in Figure 5. It implements the instructions of lines 2-9 in systolic Montgomery algorithm of Figure 3.

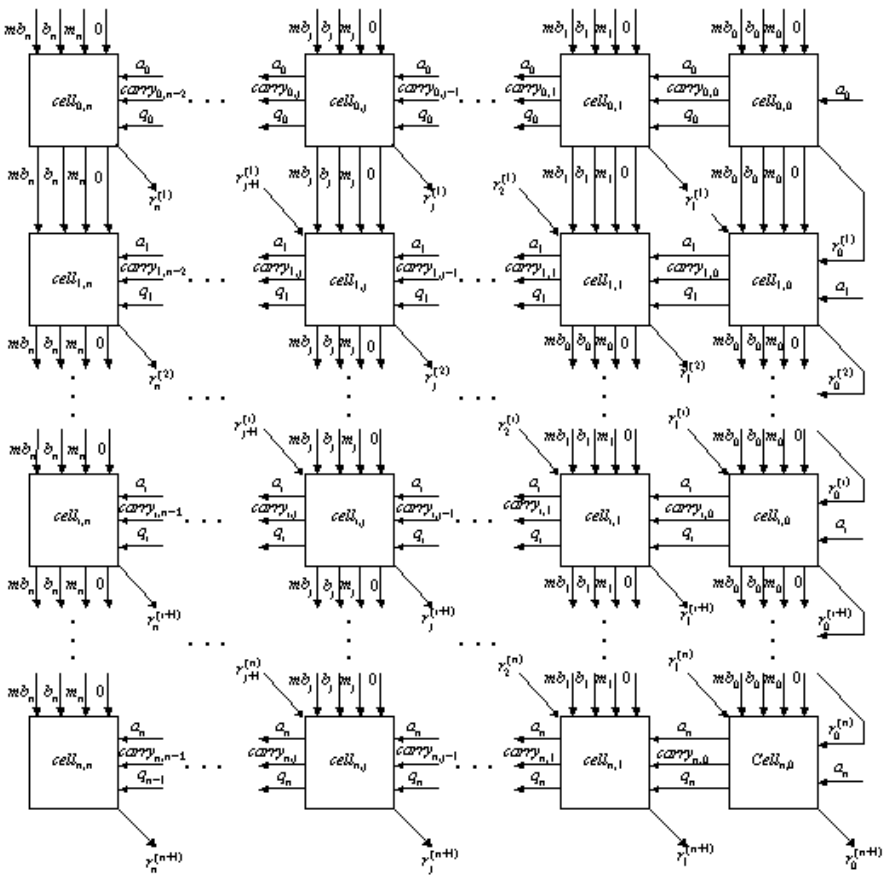

Fig. 4. Systolic architecture of Montgomery modular multiplier

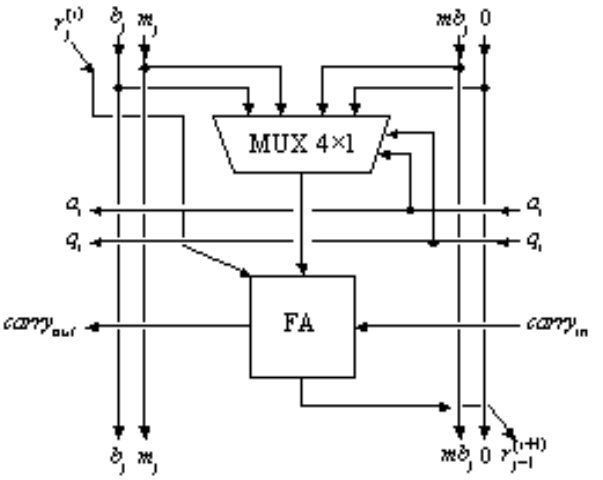

Fig. 5. Basic PE architecture

The architecture of the right-most top-most PE, i.e. $cell_{0,0}$, is given in Figure 6. Besides the computation of lines 2-9, it implements the computation indicated in line 1. However as $r_0^{(0)}$ is zero, the computation of q_0 is reduced to $a_0 \,.\, b_0$. Besides, the full-adder is not necessary as carry in signal is also 0 so $r_1^{(0)} \oplus x_i \oplus carry$ and $r_1^{(0)} \,.\, x_i + r_1^{(0)} \,.\, carry + x_i \,.\, carry$ are reduced to x_i and 0.

The architecture of the rest of the PEs of the first column is shown in Figure 7. It computes q_0 in the more general case, i.e. when $r_0^{(i)}$ is not null. Moreover, the full-adder is substituted by a half-adder as the carry in signals are zero for these PEs.

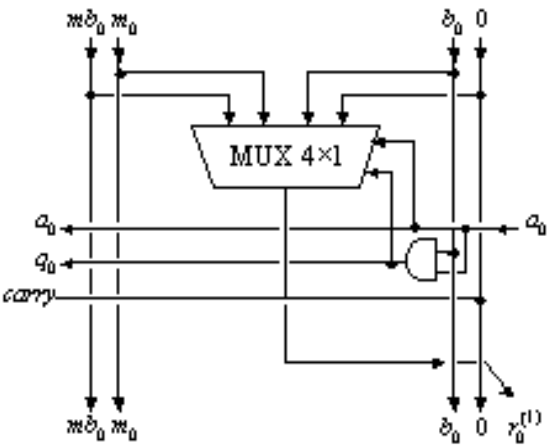

Fig. 6. Right-most top-most PE – $cell_{0,0}$

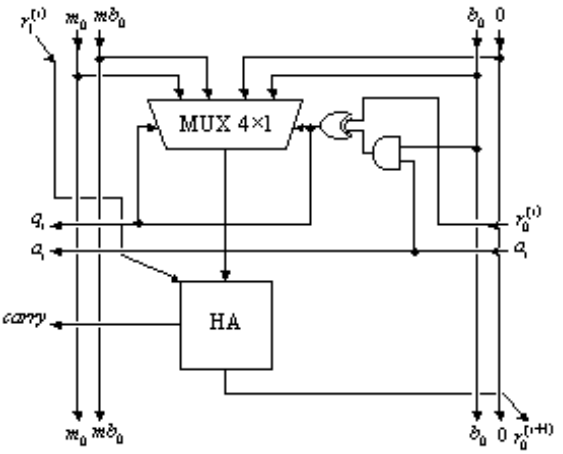

Fig. 7. Right border PEs – $cell_{i,0}$

The architecture of the architecture of the left border PEs, i.e. $cell_{0,j}$, is given in Figure 8. As $r_n^{(i)} = 0$, the full-adder is unnecessary and so it is substituted by a half-adder.

The sum $M+B$ is computed only once at the beginning of the multiplication process. This is done by a row of full adder as depicted in Figure 9.

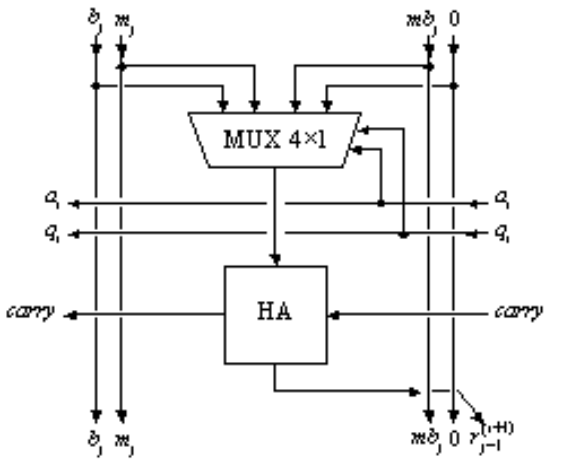

Fig. 8. Left border PEs – $cell_{0,j}$

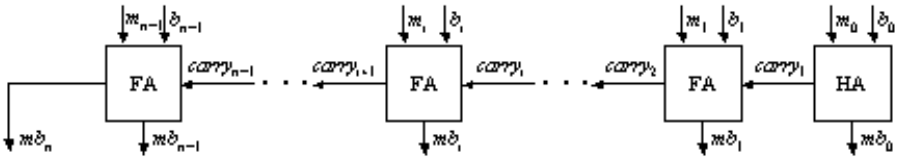

Fig. 9. Top row of the systolic multiplier

4 Systolic Binary Modular Exponentiation

The binary method scans the bits of the exponent either from left to right or from right to left. A modular squaring is performed at each step, and depending on the scanned bit value, a subsequent modular multiplication is performed. We prefer to use the left-to-right binary exponentiation method as the right-to-left method needs and extra variable (i.e. an extra register) to bookkeeping the powers of the text to be encrypted/decrypted (for details see [13]). There other more sophisticated methods to compute modular exponentiation. A comprehensive list of such methods can be found in Gordon's survey [14]. These methods, however, are more suitable for software implementations rather that hardware ones as they need a lot of memory space to store results of necessary pre-computations as well as an extensive pre-processing of the exponent before starting the actual exponentiation process.

Let k be the number of bits in the exponent. We assume that the most significant bit of the exponent is 1. The algorithm of the left-to-right binary exponentiation method is given in Figure 10.

The systolic linear architecture of the left-to-right binary modular exponentiator is given in Figure 11. It uses k–1 *e-PEs* to implement the iteration of line 1 (in algorithm *BinaryExpMethod* of Figure 10). Each of these PEs consists of two systolic modular multipliers, which are described in Section 3. The first one performs the the modular squaring of line 2 while the second one perfoms the modula multiplication of line 4 of the same algorithm. The second operand of this modular multiplication is either 1 or the text to be encrypted/decrypted, depending on the value of the bit (from the exponent) provided to the *e-PE*. The condition of line 3 is implemented by a multiplexer that passes either the text or 1.

```
algorithm BinaryExpMethod(text, modulus, exponent)
  0: cipher = text;
  1: for i = k-2 downto 0
  2:         cipher = cipher^2 mod modulus;
  3:         if exponent_i = 1 then
  4:                 cipher = cipher×text mod modulus;
return cipher;
```

Fig. 10. Left-to-right binary exponentiation algorithm

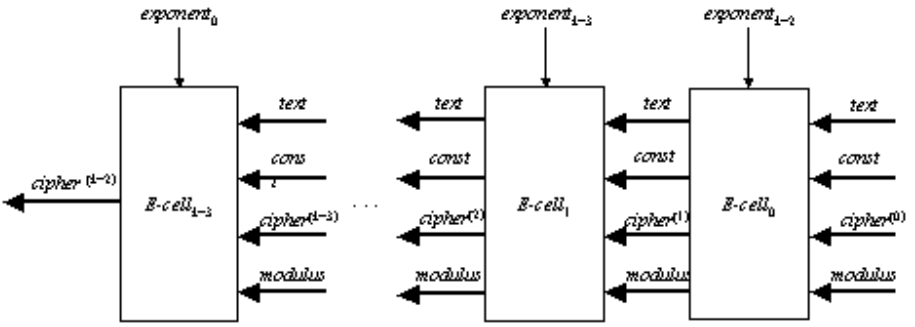

Fig. 11. The systolic linear architecture of the binary modular exponentiator

The architecture of an *e-PEs* is shown in Figure 12 wherein *SAMMM* stands for *s*ystolic *a*rray *M*ontgomery *m*odular *m*ultiplier.

The Montgomery algorithm (of Figure 1, Figure 2 and Figure 3) yields the result $2^{-n}\times A\times B$ mod M. To compute the right result, we need to further Montgomery multiply the result by the constant 2^n mod M. However, as we are interested rather in the exponentiation result than a simple product, we only need to pre-Montgomery multiply the operands by 2^{2n} mod M and post-Montgomery multiply the obtained result by 1 to get rid of the factor 2^n that is carried by every partial result. So the final architecture of the Montgomery exponentiator is that of Figure 12 augmented with two extra *SAMMM* PEs. The first PE performs the Montgomery multiplication $2^{2n}\times$*text* mod *modulus* and the second performs the Montgomery multiplication of 1 and the ciphertext yield by the binary method. This architecture is shown in Figure 13, wherein *SLE* stands for *s*ystolic *l*inear *e*xponentiator, i.e. that of Figure 11, two^{2n}, 2^n*text*, 2^n*cipher*, two^n, and *one* represent 2^n mod *modulus*, *text*$\times two^{2n}$ mod M, $2^n\times text^e$ mod *modulus*, $10\overset{n-1}{\ldots}0$ and $0\overset{n-1}{\ldots}01$ respectively.

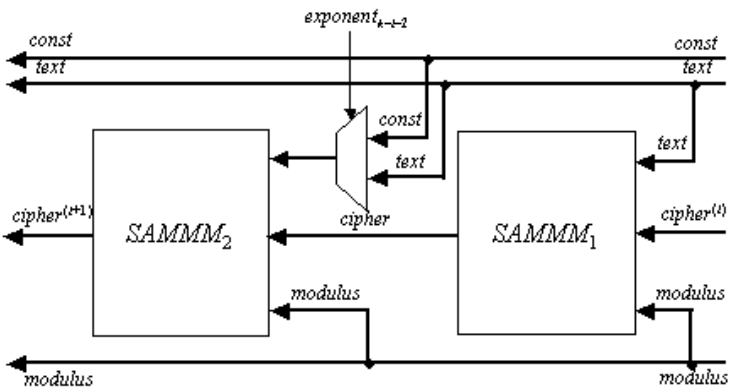

Fig. 12. Architecture of the exponentiator PE

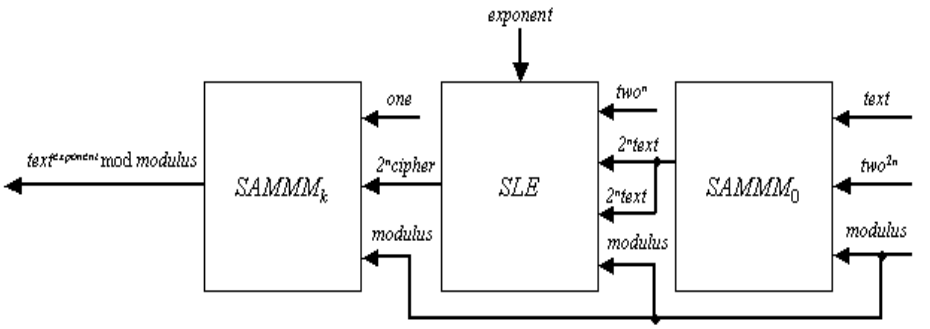

Fig. 13. Final architecture of the exponentiator

5 Time and Area Requirements

The entire design was done using the Xilinx Project Manager (version Build 6.00.09) [15] through the steps of the Xilinx design cycle shown in Figure 14. The design was elaborated using VHDL [16]. The *synthesis* step generates an optimised netlist that is the mapping of the gate-level design into the Xilinx format: *XNF*. Then, the *simulation* step consists of verifying the functionality of the elaborated design. The *implementation* step consists of partitioning the design into logic blocks, then finding a near optimal placement of each block and finally selecting the interconnect routing for a specific device family. This step generates a logic PE array file from which a bit stream can be obtained. The implementation step provides also the number of configurable logic blocks (CLBs). The *verification* step allows us to verify once again the functionality of the design and determine the response time of the design including all the delays of the physical net and padding. The *programming* step consists of loading the generated bit stream into the physical device.

The output bit $r_j^{(n+1)}$ of the modular multiplication is yield after $2n+2+j$ after bits b_j, m_j and mb_j are fed into the systolic array plus an extra clock cycle, which is needed to obtain the bit mb_j. So the first output bit appears after $2n + 3$ clock cycles. A full modular exponentiation is completed in $2k(2n + 3)$ clock cycles, including the pre- and post-modular multiplications and where k is the number of bits in the exponent. Table 2 shows the performance figures obtained by the Xilinx project synthesiser. Here we consider that the exponent has the same size as the text to encrypt/decrypt. The synthesis was done for the VIRTEX-E family.

The clock cycle time required, the area, i.e. the number of CLBs necessary as well as the time/area product delivered by the synthesis and the verification tools of the Xilinx project manager are shown in the table of Table 2.

In Blum's and Paar's work [6], only the multiplication is implemented using a systolic array. The processing element of their architecture is more complex than our multiplier PE. However, their implementation uses a single row of processing elements that are reused to accomplish an n-bit modular multiplication. So Blum and Paar reduce the required hardware area at the expense of response time as they have to include a synchronised in-control. The implementation of the exponentiation

operation is not a systolic implementation, which should again reduce area at the expense of response time. In the contrary to that of Blum's and Paar's, our implementation attempts to minimise time requirements at the expense of hardware area as we think that one can afford hardware area if one can gain in encryption/decryption time. The clock cycle time required, the area, i.e. the number of CLBs necessary as well as the time/area product delivered by Blum and Paar [7] project manager are shown in the table of Table 3.

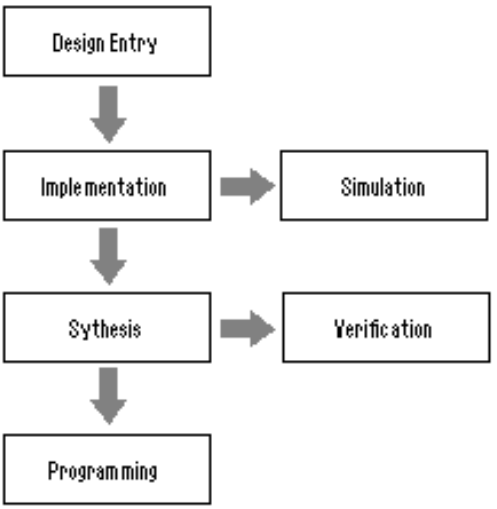

Fig. 14. Design cycle

Table 2. Performance figures of our implementation

operand size	*Area* (CLBs)	clock cycle time (ns)	area×time
128	3179	3.3	10490
256	4004	4.2	26426
512	5122	5.1	36366
768	6278	5.9	52107
1024	7739	6.6	68877

Table 3. Performance figures of Blum's and Paar's implementation

operand size	*Area* (CLBs)	clock cycle time (ns)	area×time
256	1180	19.7	23246
512	2217	19.5	43231
768	3275	20.0	65500
1024	4292	22.1	8039

The chart of Figure 15 compares the area/time product of Blum's and Paar's implementation vs. our implementation. It shows that our implementation improves the product as well as time requirement while theirs improves area at the expense of both time requirement and the product.

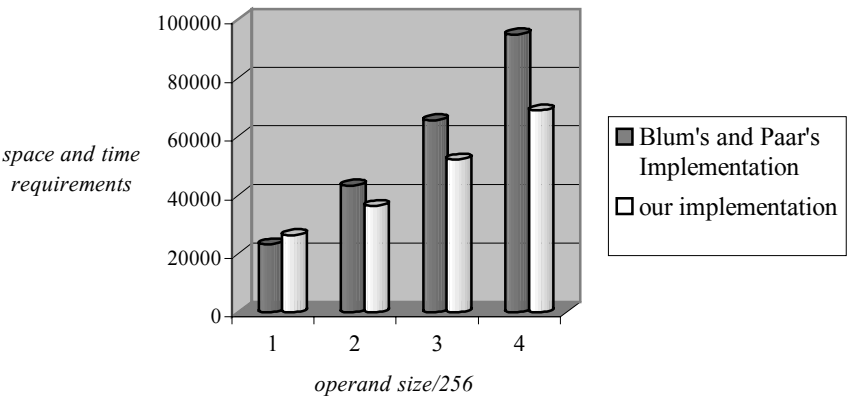

Fig. 15. Comparison of space and time requirements

6 Conclusion

In this paper, we described a systolic architecture of the modular multiplier. Then we engineered a novel systolic architecture for the exponentiator, which based on the binary exponentiation method. We compared the space and time requirements of our implementation, produced by the Xilinx project manager against those produced for Blum's and Paar's implementation. The results show clearly that despite of requiring much more hardware area, our implementation improves substantially the time requirement and the area/time product when the operand size is bigger than 256 bits. This is almost always the case in RSA encryption/decryption systems.

The authors' related work can be found in [17] and [18]. In [17] we compare a parallel implementation of the Montgomery modular multiplier vs. a sequential one and in [18] an alternative parallel binary exponentiator is described and evaluated in terms of area and performance.

References

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

[2] E. F. Brickell, *A survey of hardware implementation of RSA*, In G. Brassard, ed., Advances in Crypltology, Proceedings of CRYPTO'98, Lecture Notes in Computer Science **435**:368-370, Springer-Verlag, 1989.

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

[4] C. D. Walter, *An improved linear systolic array for fast modular exponentaition*, IEE Computers and Digital Techniques, **147**(5):323-328, 2000.

[5] A. Tiountchik, *Systolic modular exponentiation via Montgomery algorithm*, Electronic Letters, **34**(9):874-875, 1998.

[6] K. Iwamura, T. Matsumoto and H. Imai, *Montgomery modular multiplictaion and systolic arrays suitable for modular exponentiation*, Electronics and Communications in Japan, **77**(3):40-51, 1994.

[7] T. Blum and C. Paar, *Montgmery modular exponentiation on econfigurable hardware,* 14th IEEE Symposium on Computer Arithmetic, April 14-16, 1999, Adelaide, Australia.

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

[9] N. Nedjah and L. M. Mourelle, *Yet another implementation of modular multiplication*, Proceedings of 13th. Symposium of Computer Architecture and High Performance Computing, IFIP, Brasilia, Brazil, pp. 70-75, September 2001.

[10] N. Nedjah and L. M. Mourelle, *Simulation model for hardware implementation of modular multiplication*, Proceedings of WSES/IEEE International. Conference on Simulation, Knights Island, Malta, September 2001.

[11] P.L. Montgomery, *Modular multiplication without trial division*, Mathematics of Computation 44, pp. 519-521, 1985.

[12] L. M. Mourelle and N. Nedjah, *Reduced hardware Architecture for the Montgomery modular multiplication*, Proceedings of the third WSEAS Transactions on Systems, **1**(1):63-67, January 2002.

[13] Ç.K. Koç, *High-speed RSA implementation*, Technical report, RSA Laboratories, RSA Data Security, Inc. Redwood City, CA, USA, November 1994.

[14] D. M. Gordon, *A survey of Fast exponentiation methods*, Journal of Algorithms 27, pp. 129-146, 1998

[15] Xilinx, Inc. *Foundation Series Software*, Http://www.xilinx.com.

[16] Z. Navabi, *VHDL - Analysis and modeling of digital systems*, McGraw Hill, Second Edition, 1998.

[17] N. Nedjah and L. M. Mourelle, *Two Hardware Implementations for the Montgomery Modular Multiplication: Parallel vs. Sequential*, Proceedings of 15th. Symposium on Integrated Circuits and Systems Design, IEEE Computer Society, Porto Alegre, Brazil, pp. 3-8, September 2002.

[18] N. Nedjah and L. M. Mourelle, *Reconfigurable Hardware Implementation of Montgomery Modular Multiplication and Parallel Binary Exponentiation*, Proceedings of Euromicro Symposium on Digital Systems Design, IEEE Computer Society, Dortmund, Germany, pp. 226-233, September 2002.