Two Hardware Implementations for the Montgomery Modular Multiplication: Sequential versus Parallel

Nadia Nedjah and Luiza de Macedo Mourelle
Department of de Systems Engineering and Computation,
Faculty of Engineering,
State University of Rio de Janeiro
(ldmm | nadia)@eng.uerj.br

Abstract

Modular multiplication is the most dominant part of the computation performed in public-key cryptography systems such as the RSA cryptosystem. The operation is time consuming for large operands. This paper describes the characteristics of two architectures designed to implement modular multiplication using the fast Montgomery algorithm: the first FPGA prototype has an iterative sequential architecture while the second has a systolic array-based architecture. The paper compares both prototypes using the time×area classic factor.

1. Introduction

An RSA cryptosystem consists of a set of three items: a modulus M and two integers d and e called private and public keys respectively that satisfy the property $T^{DE} = T$ mod M with plain text T obeying $0 \leq T < M$. Messages are encrypted using the public key as $C = T^D$ mod M and decrypted as $T = C^E$ mod M. So the same operation, i.e. modular exponentiation, is used to perform both processes: encryption and decryption. It consists of a repetition of modular multiplications. Hardware implementation of the RSA cryptosystem is widely studied as in [1], [2], [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 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.

There are various algorithms that implement modular multiplication such as Barrett's and Booth's methods [12], [13], and Brickell's algorithm [14]. Here, we concentrate on the Montgomery's algorithm as it is considered the most popular and more efficient.

In this paper, we present two different prototypes for implementing the Montgomery's modular multiplication: One has a sequential architecture and the other a fully parallel architecture. The first prototype reduces area usage in detriment of response time while on the contrary the second one reduces time response in detriment of area requirement. However the parallel prototype improves the area/time product.

The rest of this paper is organised as follows: in Section 2, we describe the Montgomery's algorithm used to implement the modular operation; in Section 3, we present the architecture of the iterative hardware of the Montgomery's modular multiplier; in Section 4, we modify the Montgomery's algorithm to highlight the systolic nature of the computation and propose the architecture of the systolic prototype; finally, in Section 5, we compare both prototypes in terms of area and time requirements.

2. The Montgomery's 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 P modulo 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, where n is the size of the operands [5].

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

IEEE COMPUTER SOCIETY

product generators together with several adders that operate in parallel [6], [7], [8].

One of the widely used algorithms for efficient modular multiplication is the Montgomery's algorithm [9]. This algorithm computes the product of two integers modulo a third one without performing division by M. 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 digits 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;*
- *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's 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×M is a multiple of the radix r, and this is right-shifted by r positions, i.e. divided by r for use in the next iteration. So, after n iterations, the result obtained is $R = A \times B \times r^{-n}$ mod M. A modified version of the Montgomery's algorithm is given in Fig.1.

```
algorithm Montgomery(A, B, M) {
  int R = 0;
1: for i= 0 to n-1 {
2:   R = R + $a_i \times B$;
3:   if $r_0$ = 0 then
4:     R = R div 2
5:   else
6:     R = (R + M) div 2;
  }
  return R;
}
```

Figure 1: Montgomery modular algorithm.

In order to yield the right result, we need an extra Montgomery modular multiplication by the constant 2^n mod M. However as the main objective of the use of Montgomery modular multiplication algorithm is to compute exponentiations, it is preferable to Montgomery pre-multiply the operands by 2^{2n} and Montgomery post-multiply the result by 1 to get rid of the 2^{-n} factor. Here we concentrate on the implementation of the Montgomery multiplication algorithm of Fig.1.

3. Iterative Prototype For Montgomery Multiplication

In this section, we outline the architecture of the Montgomery modular multiplier. Its interface consists of the operands A, B and M as inputs and $R = (A \times B \times 2^{-n})$ mod M as output.

The detailed architecture of the Montgomery modular multiplier is given in Fig.2. *It uses two multiplexers, two adders, two shift registers, three registers as well as a controller. The latter will be described in the next section.*

The first multiplexer of the proposed architecture, i.e. $MUX2_1$, passes 0 or the content of register B depending on whether bit a_0 indicates 0 or 1 respectively. The second multiplexer, i.e. $MUX2_2$ passes 0 or the content of register M depending on whether bit r_0 indicates 0 or 1 respectively. The first adder, i.e. $ADDER_1$, delivers the sum $R + a_i \times B$ (line 2 of algorithm of Figure 1), and the second adder, i.e. $ADDER_2$, yields the sum R + M (line 6 of the same algorithm). The shift register $SHIFT\ REGISTER_1$ provides the bit a_i. At each iteration i of the multiplier, this shift register is right-shifted once, so that the least significant bit of $SHIFT\ REGISTER_1$ contains a_i.

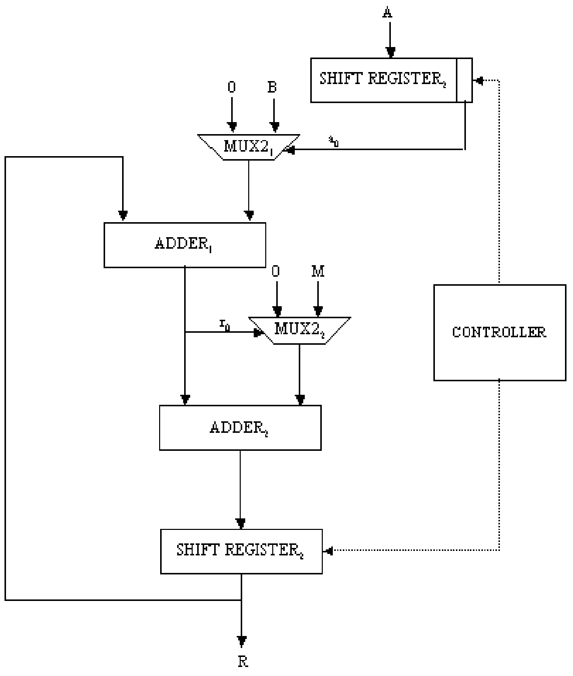

Figure 2: Iterative Montgomery multiplier architecture.

The role of the controller consists of synchronising the shifting and loading operations of SHIFT REGISTER$_1$ *and* SHIFT REGISTER$_2$. *It also controls the number of iterations that have to be performed by the multiplier. For this purpose, the controller uses a simple down counter. The counter is inherent to the controller. The interface of the controller is given in* Fig.3.

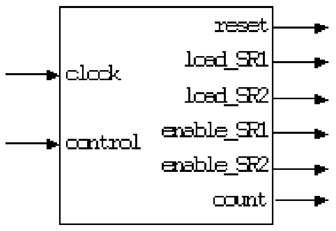

Figure 3: Interface of the multiplier controller.

In order to synchronise the work of the components of the architecture, the controller is implemented by a state machine, which has 6 states defined as follows:

- *S_0: initialisation of the state machine;*
  go to S_1;
- *S_1: load multiplicand and modulus into registers;*
  load multiplicator into SHIFT REGISTER$_1$;
  go to S_2;
- *S_2: wait for* ADDER$_1$;
  wait for ADDER$_2$;
  load multiplicator into SHIFT REGISTER$_2$;
  increment counter;
  go to S_3;
- *S_3: enable* SHIFT REGISTER$_2$;
  enable SHIFT REGISTER$_1$;
- *S_4: check the counter;*
  if 0 then go to S_5 else go to S_2;
- *S_5: halt;*

4. Systolic Prototype For the Montgomery's Multiplication

A modified version of the Montgomery's algorithm is that of Fig. 5. *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 Fig.4. *It can be computed as indicated in the last column of the table of* Fig. 5 *depending on the value of the bits a_i and q_i.*

```
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;
}
```

Figure 4: Modified Montgomery algorithm.

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

Figure 5: Computation of $R + a_i \times B + q_i \times M$.

A bit-wise version of the algorithm of Fig. 4, *which is at the basis of our systolic implementation, is described in* Fig. 6. *All algorithms, i.e. those of* Fig. 1, Fig. 4 *and* Fig. 6 *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 Fig. 6 *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.

```
algorithm   SystolicMontgomery(A,B,M,MB)
{
  int R ← 0; bit carry ← 0, x;
  0: for i = 0 to n {
  1:    $q_i \leftarrow r_0^{(i)} \oplus a_i.b_0$;
  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:       $r_j^{(i+1)} \leftarrow r_{j+1}^{(i)} \oplus x_i \oplus$ carry;
  9:       carry ← $r_{j+1}^{(i)}.x_i + r_{j+1}^{(i)}$.carry + $x_i$.carry;
       }
    }
  return R;
}
```

Figure 6: Systolic Montgomery modular algorithm.

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 systolic Montgomery multiplier is shown in Fig. 7.

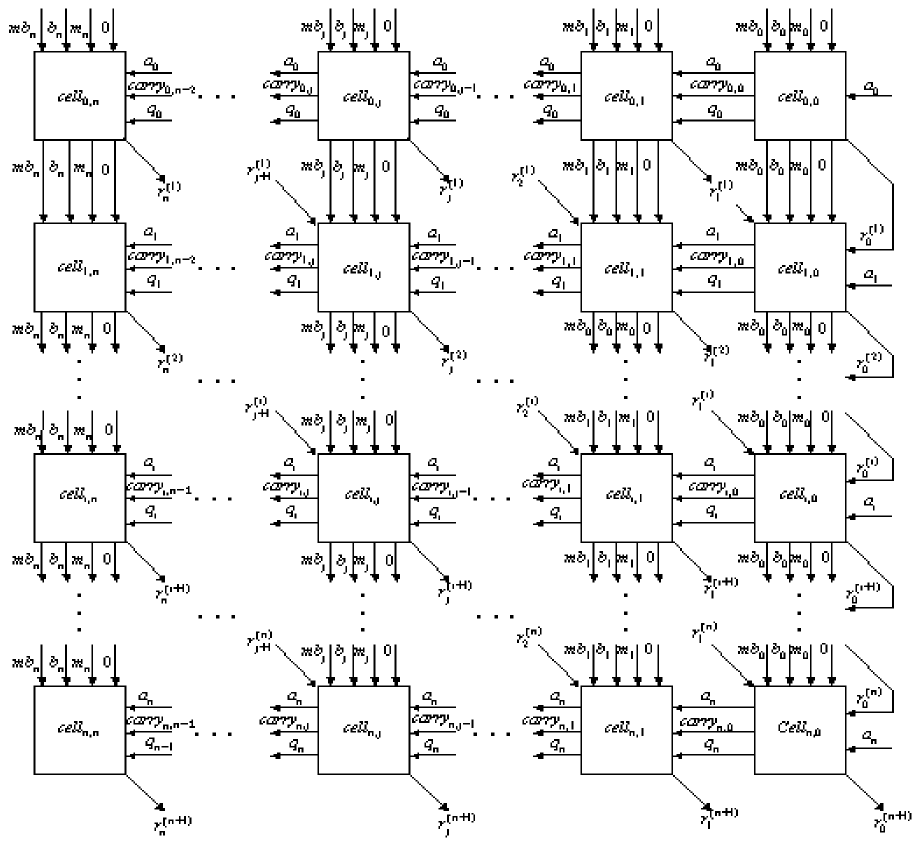

Figure 7: Systolic architecture of Montgomery multiplier.

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 Fig. 8. *It implements the instructions of lines 2-9 in systolic Montgomery algorithm of* Fig.6.

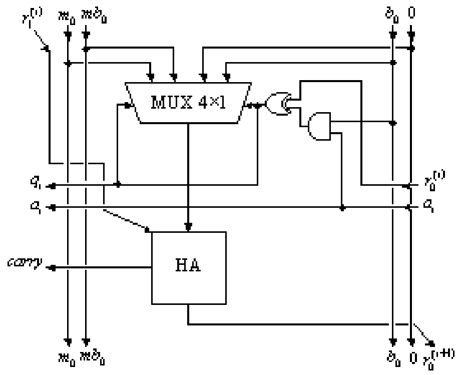

Figure 8: Right border PEs – cell$_{i,0}$.

The architecture of the architecture of the left border PEs, i.e. cell$_{0,j}$, is given in Fig. 9. *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 adders.*

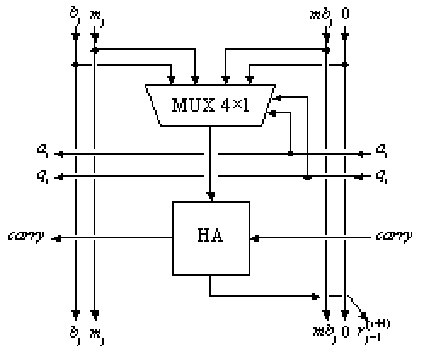

Figure 9: Left border PEs – cell$_{0,j}$.

5. Time and Area Requirements

The entire design was done using the Xilinx Project Manager (version Build 6.00.09) [14] through the steps of the Xilinx design cycle shown in Fig. 10. *The design was elaborated using VHDL [15]. 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.*

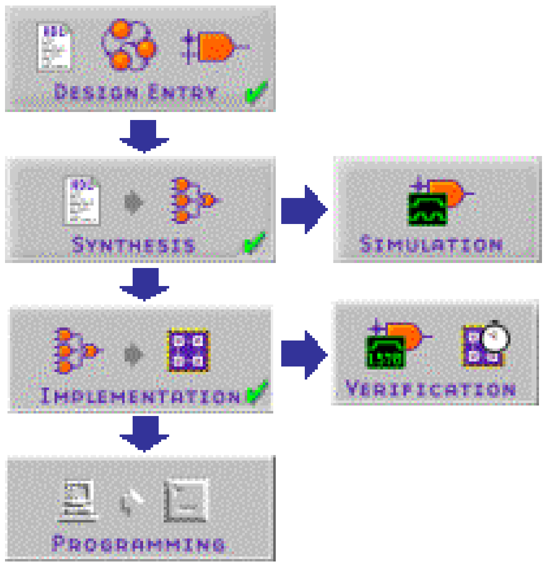

Figure 10: *Design cycle*.

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.* Fig. 11 *shows the performance figures obtained by the Xilinx project synthesiser for the iterative multiplier the systolic modular multiplier, wherein IM and SM stand for iterative multiplier and systolic multiplier respectively. The synthesis was done for VIRTEX-E family.*

The table of Fig. 11 *shows 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 for the iterative and systolic version of Montgomery multiplier.*

operand size	***Area** (CLBs)*		***clock cycle time** (ns)*		***area×time***	
	IM	*SM*	*IM*	*SM*	*IM*	*SM*
128	*89*	*259*	*46*	*23*	*4094*	*5957*
256	*124*	*304*	*102*	*42*	*12648*	*12767*
512	*209*	*492*	*199*	*76*	*41591*	*37392*

768	335	578	207	82	69345	47396
1024	441	639	324	134	142884	85626

Table 11: Performance figures: iterative vs. systolic Montgomery modular multiplier.

The chart of Fig. 12 *compares the area/time product of iterative multiplier implementation vs. the systolic implementation. It shows that the latter improves the product as well as time requirement while the former improves area at the expense of both time requirement and the product.*

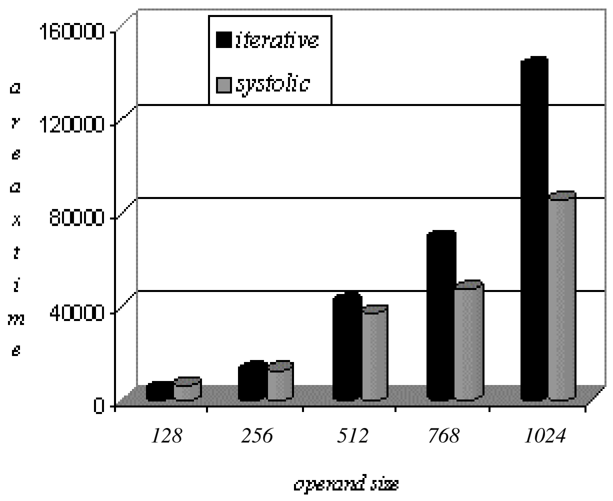

Figure 12: The areaxtime factor for iterative vs. systolic multiplier

So the iterative modular multiplier reduces the required hardware area at the expense of response time as they have to include a synchronised in-control. The systolic implementation of the modular multiplier 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.

6. Conclusion

In this paper, we described two new architectures to implement Montgomery modular multiplication algorithm: an iterative architecture and a systolic architecture.

The modular multipliers were synthesized and implemented using the Xilinx Project Manager [15]. The implementation device used is an FPGA: family SPARTAN and model S05PC84-4.

We compared the space and time requirements of both prototypes for different operand sizes, produced by the Xilinx project manager. The results show clearly that despite of requiring much more hardware area, the systolic implementation improves substantially the time requirement and the area/time product when the operand size is bigger than 512 bits, which is almost always the case in RSA encryption/decryption systems.

7. 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] *S. E. Eldridge and C. D. Walter, Hardware implementation of Montgomery's Modular Multiplication Algorithm, IEEE Transactions on Computers,* **42***(6):619-624, 1993.*

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

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

[7] N. Nedjah, L. M. Mourelle, Simulation Model for Hardware implementation of modular multiplication, Proceedings of WSES/IEEE International. Conference on Simulation, Knights Island, Malta, September 2001.

[8] N. Nedjah, L. M. Mourelle, Reduced Hardware Architecture for the Montgomery Modular Multiplication, Proceedings of the third WSEAS/IEEE Symposium Mathematical Methods and Computational Techniques in Electrical Engineering, Athens, Greece, December 2001.

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

[10] Z. Navabi, VHDL - Analysis and Modeling of Digital Systems, McGraw Hill, Second Edition, 1998.

[11] MyCad, Inc. and Seodu Logic, Inc., MyVHDL Station V 4.0 Tutorial, http://www.mycad.com or http://www.mycad.co.kr.

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

[13] G. W. Bewick, Fast multiplication algorithms and implementation, Ph. D. Thesis, Department of Electrical Engineering, Stanford University, United States of America, 1994.

[14] C. D. Walter, A verification of Brickell's fast modular multiplication algorithm, International Journal of Computer Mathematics, **33**:153:169, 1990.

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