Hardware Architecture for the Montgomery Modular Multiplication

NADIA NEDJAH AND LUIZA DE MACEDO MOURELLE
Department of de Systems Engineering and Computation,
State University of Rio de Janeiro
São Francisco Xavier, 524, 5O. Andar, Rio de Janeiro,
BRAZIL

***Abstract*:-** Modular multiplication is the most dominant part of the computation performed in public-key cryptography systems such systems. The operation is time consuming for large operands. This paper examines the characteristics of yet another architecture to implement modular multiplication using the fast Montgomery algorithm. An experimental Montgomery modular multiplier prototype is described and simulated. The simulation results are presented.

***Key-Words*:-** modular multiplication, Montgomery algorithm, simulation, cryptosystems.

1 Introduction

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, and M is called the modulus. 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 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.

An RSA cryptosystem 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 $T^{de} = T \bmod M$. Plain text T obeying $0 \leq T < M$. 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 is used to perform both processes: encryption and decryption. Hardware implementation of the RSA cryptosystem is widely studied as in [2, 3, 4].

In the rest of this paper, we start off by describing the algorithms used to implement the modular operation. Then we present the architecture of the hardware Montgomery modular multiplier and explain in details how it executes a single multiplication. Then we comment the simulation results obtained for such architecture.

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

One of the widely used algorithms for efficient modular multiplication is the Montgomery's algorithm [8]. 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 and multiplicator 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 X, Y and M as follows:

$$X = \sum_{i=0}^{n} x_i \times 2^i, \quad Y = \sum_{i=0}^{n} y_i \times 2^i \text{ and } M = \sum_{i=0}^{n} m_i \times 2^i$$

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 multiplicator 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 r positions, i.e. divided by r for use in the nest iteration. Consequently, after n iterations, the result obtained is $R = A\times B\times r^{-n} \bmod M$. A modified version of Montgomery algorithm is given in Fig. 1.

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

Fig. 1: Montgomery modular algorithm.

In order to yield the right result, we need an extra Montgomery modular multiplication by the constant $r^n \bmod M$. As we use binary representation of numbers, we compute the final result using the algorithm of Fig. 2.

```
algorithm ModularMult(A, B, M, n) {
    const C = 2^n mod M;
    int R = 0;
    R = Montgomery(A, B, M);
    return Montgomery(R, C, M);
}
```

Fig. 2: Modular multiplication algorithm.

3 Montgomery modular multiplier architecture

In this section, we outline the architecture of the Montgomery modular multiplier. The interface of the Montgomery modular multiplier is given in Fig. 3. It receives the operands A, B and M and it returns $R = (A\times B\times 2^{-n}) \bmod M$.

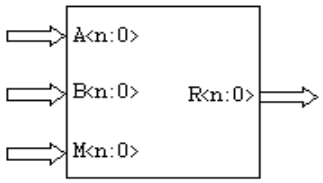

Fig. 3: The Montgomery modular multiplier interface.

The detailed architecture of the Montgomery modular multiplier is given in Fig. 4. 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. Each iteration i of the multiplier, this shift register is right-shifted once so that a_0 contains a_i.

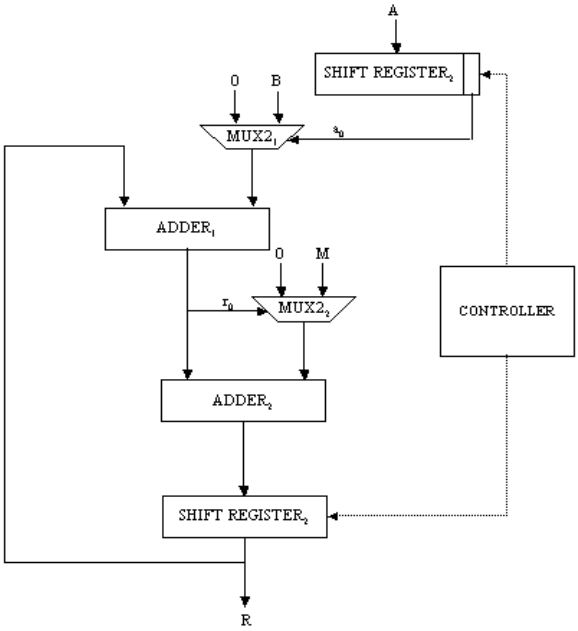

Fig. 4: The Montgomery modular multiplier architecture.

The role of the controller consists of synchronising the shifting and loading operations of the SHIFT REGISTER$_1$ and SHIFT REGISTER$_2$. It also controls the number of iterations that have to be performed by the multiplier. For this end, the controller uses a simple down counter. The counter is inherent to the controller. The interface of the controller is given in Fig. 5.

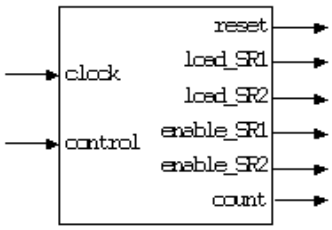

Fig. 5: The interface of the Montgomery multiplier controller

In order to synchronise the work of the components of the architecture, the controller consists of 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 Modular multiplier architecture

The modular multiplier yields the actual value of $A \times B \bmod M$. It first computes $R = A \times B \times 2^{-n} \bmod M$ using the Montgomery modular multiplier. Then, it computes $R \times C \bmod M$, where $C = 2^n \bmod M$. The modular multiplier interface is shown in Fig. 6.

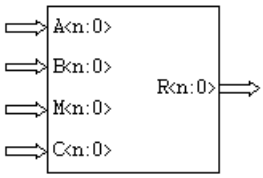

Fig. 6: The modular multiplier interface.

The modular multiplier uses a 4-to-1 multiplixer MUX4 and a register REGISTER.

- Step 0: Multiplexer MUX4 passes 0 or *B*. MUX2 passes *A*. It yields $R = A \times B \times 2^{-n} \bmod M$. Register REGISTER contains 0.
- Step 1: Multiplexer MUX4 passes 0 or *R*. MUX2 passes *C*. It yields $R = R \times C \bmod M$. Register REGISTER contains the result of the first step computation, i.e. $R = A \times B \times 2^{-n} \bmod M$.

The modular multiplier architecture is given in Fig. 7.

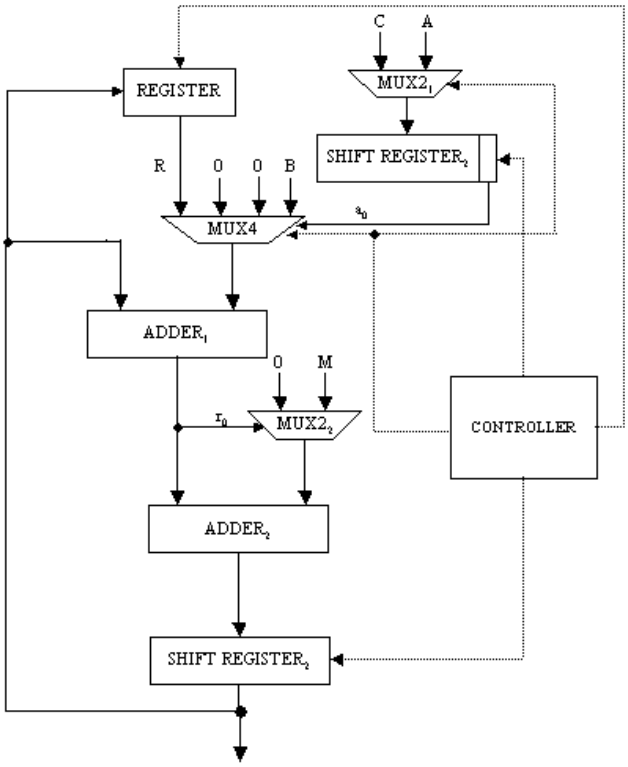

Fig. 7: The modular multiplier architecture.

In order to synchronise the work of the components of the modular multiplier, the architecture contains a controller, which consists of a state machine of 10 states. The interface of the component CONTROLLER is given in Fig. 8.

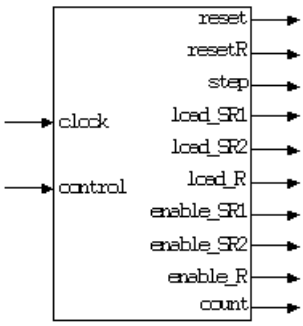

Fig. 8: The interface of the modular multiplier controller

The modular multiplier controller does all the control that the Montgomery modular multiplier needs as described in Section 3. Furthermore, it controls the changing from step 0 to step 1, the loading of the register REGISTER.

- S_0: initialisation of the state machine;
  go to S_1; set step 0;
- 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 partial result into *shift register*$_2$;
  increment counter;
  go to S_3;
- S_3: enable shift register$_2$;
  enable shift register$_1$;
  go to S_4;
- S_4: load the partial result of step 0 into *register*;
  check the counter;
  if 0 then go to S_5 else go to S_2;
- S_5: load constante into *shift register*$_1$;
  reset *register*;
  set step 1;
  go to S_6;
- S_6: wait for *ADDER*$_1$;
  wait for *ADDER*$_2$;
  load partial result into *shift register*$_2$;
  increment counter;
  go to S_7;
- S_7: enable shift register$_2$;
  enable shift register$_1$;
  go to S_8;
- S_8: check the counter;
  if 0 then go to S_9 else go to S_6;
- S_9: halt.

5 Conclusion

In this paper, an alternative architecture for computing modular multiplication based Montgomery algorithm is described. Other algorithms exist, such as Barrett's and Booth's method [11, 12], and Brickell's algorithm [13].

The project of the modular multiplier described throughout this paper was specified in Very High Speed Integrated Circuit Description Language - VHDL [9], and simulated using *the MyVHDL Station* of *MyCad* Inc. [10].

The prototype of the modular multiplier was then synthesized and implemented using the Xilinx Project Manager [14]. The implementation device used is an FPGA: family SPARTAN and model S05PC84-4. The area needed to implement the prototype is about 624 equivalent gates and the maximum net delay is about 41 ns, with a maximum period of about 18ns, i.e. maximum frequency of 58 MHz.

6 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 International. Conference on Simulation, WSES, Knights Island, Malta, September 2001.

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

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

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

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

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

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

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