High Radix Implementation of Montgomery Multipliers with CSA

Gashaw Sassaw
Instituto de Microelectrónica de Sevilla
Sevilla, Spain
Email: sassaw@imse-cnm.csic.es

Carlos J. Jiménez
University of Seville / Instituto de Microelectrónica de Sevilla
Sevilla, Spain
Email: cjesus@imse-cnm.csic.es

Manuel Valencia
University of Seville / Instituto de Microelectrónica de Sevilla
Sevilla, Spain
Email: manolov@imse-cnm.csic.es

[1]

***Abstract*— Modular multiplication is the key operation in systems based on public key encryption, both for RSA and elliptic curve (ECC) systems. High performance hardware implementations of RSA and ECC systems use the Montgomery algorithm for modular multiplication, since it allows results to be obtained without performing the division operation. The aim of this article is to explore various modified structures of the Montgomery algorithm for high speed implementation. We present the implementation of a modified Montgomery algorithm with CSA and with different radix. In order to optimize the implementation regarding operation speed, we considered carry save adders structures and the Booth recoding scheme. The structure used in this paper simplifies the computation of the partial products avoiding the use of memories to store pre-calculated data for partial products which cannot be achieved by the shifting operation. The result shows that high-radix implementations are better for high speed applications.**

***Index Terms*— Hardware implementation, Modular multipliers, Montgomery Multipliers, Public key cryptosystems.**

I. Introduction

The increase in data communication and the expansion of internet services like email, e-commerce and e-banking have made cryptography an important research topic crucial for the provision of confidentiality, authentication, data integrity, and non-reputation. Unlike symmetric key cryptosystems, public-key cryptosystems are capable of fulfilling all of these objectives. The use of portable devices and the need for increased speed in some applications, means that public-key schemes should preferably be implemented in hardware.

The idea of public-key cryptosystems was originally presented by Diffie and Hellman [1]. With this scheme, two entities can communicate securely via an insecure communication channel without sharing a secret key. Today there are two alternatives for communicating with public key cryptography: RSA [2] and Elliptic Curve Cryptography (ECC) [3][4]. In RSA the main operation is to compute modular exponentiation by repeat modular multiplication. The security of RSA algorithms lies in the difficulty of factorizing large integers (e.g. 1024 bits). For elliptic-curve-based protocols, it is assumed that finding the discrete logarithm of a random elliptic curve element with respect to a publicly-known base point is not feasible. The size of the elliptic curve determines the difficulty of the problem. ECC has shorter key size than RSA. Both RSA and ECC require fast modular multiplication at a precision of 192 to 2048 bits.

Among the various algorithms presented for carrying out modular multiplications, Montgomery's algorithm is the one best suited for hardware implementations and therefore the most widely used [5]. The algorithm, presented by Peter L. Montgomery in 1985 [6] is used to speed up modular multiplication [7][8].

This paper is organized as follows: the section below briefly describes the Montgomery multiplication algorithm. The options for an efficient hardware implementation are discussed in Section III. Section IV presents the results of the hardware implementations and finally some conclusions are drawn.

II. Montgomery's Multiplication Algorithm

The most relevant feature of the Montgomery algorithm is that it avoids the division operation during modular multiplication. It only needs multiplication, addition and right shift which are easily implemented in hardware.

The modular multiplication problem is defined as the computation of $P = AB \pmod N$, where A, B and N are positive n bits numbers with $0 \leq A, B < N$. The Montgomery algorithm solves the operation as:

$$P = AB \pmod N = ABR^{-1} \pmod N$$

where R^{-1} is the inverse of R modulo N, i.e., it is a number with the property

$$R^{-1}R = 1 \bmod N$$

Montgomery's multiplication algorithm requires R and N to be relatively prime, i.e.:

[1] This work was supported in part by the European Community through the MOBY-DIC project (FP7-INFSO-ICT-248858, www.mobydic-project.eu), the Spanish Government through the TEC2007-65105/MIC project and for the Junta de Andalucía through the P08-TIC-03674 project.

$$\gcd(R,N) = \gcd(2^n,N) = 1$$

Where gcd is the "greater common divisor", and R > N. This expression is satisfied if N is odd as is required by the algorithm. In order to describe his algorithm, Montgomery introduced the quantity, N', which should satisfy $0 < N' < R$ and $RR^{-1} - NN'=1$.

The Montgomery multiplication is computed as follows:

```
Step 1: $T = AB$
Step 2: $P = (T + (TN' \bmod R)N)/R$
Step 3: if $P \geq N$ then $P = P - N$
```

This algorithm uses multiplication modulo R and division by R, which are faster and simpler than the computation of AB mod N by conventional methods involving division by N [9]. This algorithm is only efficient when performing multiple modular multiplications, such as modular exponentiation operations.

In hardware implementation of modular exponentiation, the exponentiation operation is replaced by a series of squaring and multiplication operations [10]. The expression $x = a^e$ (mod n), where e is a number of j bits, can be calculated with j calls to a modular multiplication. The steps are:

```
Step 1: $\bar{a} = a \cdot r \bmod n$
Step 2: $\bar{x} = 1 \cdot r \bmod n$
Step 3: for i=j-1 downto 0
            $x = \mathrm{MonPro}(\bar{x}, \bar{x})$
            If $e_i$= 1 then $\bar{x} = \mathrm{MonPro}(\bar{x}, \bar{a})$
Step 4: return $x = \mathrm{MonPro}(\bar{x}, 1)$
```

Step 4 of the modular exponentiation algorithm computes x using $\bar{x}$ via the property of Montgomery algorithm:

$$\mathrm{MonPro}\left(\bar{x},1\right) = \bar{x}1r^{-1} = xr^{r-1} = x \bmod n$$

III. Hardware Implementation of Montgomery Multipliers

For efficient hardware implementations of Montgomery multipliers some modifications to the original algorithm have been proposed [11]. These changes follow several lines: one of these lines is the use of graphical models for multiplier partitioning [12] [13]. Another line is the implementation of modular multipliers through partitioning, obtaining scalable architectures [14][15][16]. With this structure the modular multiplication of numbers with many bits is achieved from multipliers with smaller numbers of bits.

Another line of work is aimed at improving the performance of implementations of Montgomery multipliers by introducing carry save adders and using of high radix [17][18]. The implementations presented in this paper follow this line.

The introduction of carry-free adders helps to reduce the delay that arises because of the long carry propagation line from the adding process that has to be done in a modular exponentiation operation. Several alternatives have been proposed to reduce carry propagation, but they can be divided into two categories. In the first category, intermediate results are kept in redundant form with the help of carry save adders. However, at the end of each modular multiplication the output needs to be converted to a conventional binary number [19][20]. In the second category, Montgomery multiplier inputs and outputs are also kept in redundant format. In a modular exponentiation operation the Montgomery algorithm is executed several hundred times, so the latter option is more efficient [12]-[19].

Furthermore, implementations of Montgomery multipliers with high-radix reduce the number of iterations required to obtain each result. The combination of carry save adders with high-radix implementations increases operating speed but consumes more resources.

We have implemented Montgomery multipliers with radix-2, radix-4, radix-8 and radix-64. All implementations considered CSA adders, where both the intermediate data and the inputs and outputs are kept in redundant format. In addition to the CSA adders, high-radix structures are implemented using the Booth algorithm, which simplifies the accumulation of partial products.

We started implementing the radix-2 Montgomery multiplier with CSA. There are two alternatives for the radix-2 CSA adder structure: five-to-two and four-to-two [15]. In this paper we used the five-to-two structure, because its simpler control system makes it suitable for high-radix implementations. The design methodology for Booth encoded Montgomery modules based on ripple carry adders is presented in [20]. Figure 1 illustrates the implementation of the Montgomery algorithm stated in section II, but it was modified somewhat with regard to the accumulation of the partial product and zeroing factor (parity; which ensures that the least significative bit is zero before the shifting process). In the figure the control circuits are not shown. As can be seen in Figure 1, this structure prevents the propagation of carry in the intermediate operations and also between successive operations of the Montgomery algorithm, since the inputs and outputs are in redundant format.

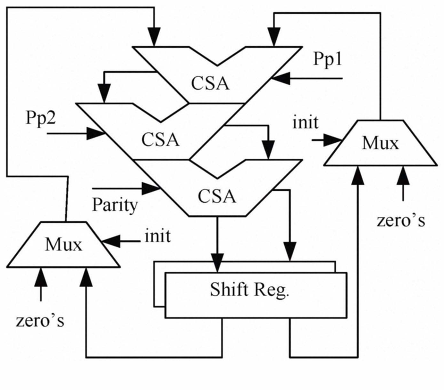

Fig. 1: five-to-two structure for Mongomery Algorithm

As already mentioned, the implementation of high-radix multipliers increases circuit complexity, and therefore reduces the maximum operation frequency. The main problem is obtaining a partial product of the multiplicand (B) and the module (M), which cannot immediately be done through shifting. In most implementations, the partial product (multiples) is generated with ripple carry adders, using memories or registers for the storage of pre-computed values [21]. However, in the implementations presented here, the multiple that cannot be obtained by shifting is achieved directly through the carry save adders structure. The application of this technique is simplified when it is applied in conjunction with the Booth recoding scheme [18].

In the case of radix-8 multipliers, with the Booth recoding scheme the set of values that multiply the multiplicand and the module are (-4, -3, -2, -1, 0, 1, 2, 3, 4). Within the set the multiple factors 3 and -3 are not achieved without an arithmetic operation. In the implementations proposed in this paper, both values are obtained with the sums 2 + 1 and -1 + -2 respectively. The multipliers include a CSA adder structure capable of performing these calculations without pre-calculating and storing the value.

In the case of radix-64 implementations, of the 32 combinations of values for the multiple of the multiplicand and the module, only 11 can be calculated through a shifting operation. The remaining 20 must be calculated using the available CSA structure. The Booth encoding scheme is used for generating multiples of both the multiplicand and the module. The purpose of the multiple of the module is to prevent data loss during right shifting.

Figure 2 illustrates all the partial products that must be generated for a radix-64 implementation. These partial products are divided into three groups. In each clock cycle three partial products are generated (one in each group). These partial products are applied for both the multiplicand and the module.

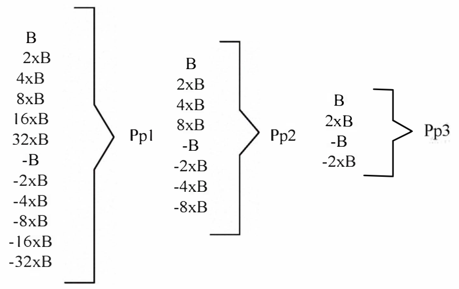

Fig. 2: Partial Product for radix-64

In each clock cycle it will therefore be necessary to accumulate a maximum of three partial products for both the multiplicand and the module, and the outputs of the accumulation registers. This requires an eight-to-two structure as is shown in Figure 3. In this CSA structure there are six inputs available for multiples, 3 for the multiplicand (B) and 3 for the module (M).

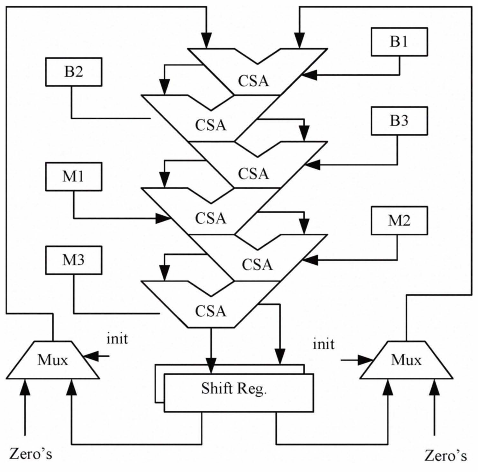

Fig. 3: Eight-to-two CSA structure for radix-64 Montgomery Algorithm.

IV. Results

Implementation was started by describing the design in VHDL language. The VHDL description is parametrizable, meaning that the number of bits of the inputs and outputs was determined at the beginning of the synthesis process. The VHDL description also fulfils the synthesis tool requirement. Once the logical functionality of the design was verified, we proceeded to the implementation, for which we used two technologies, one FPGA and one ASIC.

For FPGA implementations the Xilinx device XC2V6000 was used. Table I shows the results of the occupied area for the different implementations. The number of slices grows with the increase in the base.

TABLE I
AREA SIZE AND CLOCK CYCLES FOR FPGA IMPLEMENTATIONS.

Slice	Radix-2	Radix-4	Radix-8	Radix-64
Occupied	10,683	13,535	29,459	33,187
Clock cycles	1024	512	256	32

For ASIC implementations a CMOS 130 nm technology was used. The design and characterization methodology followed was based on a semicustom design flow and the use of commercial tools, for both front-end and back-end. The characterization methodology followed in this study was the one proposed in [22]. The characterization results are shown in Table II. In this table it can be seen that increasing the radix increases the consumption of resources, thus increasing the area occupied by the circuit and reducing the maximum frequency.

TABLE II
AREA, DELAY AND FREQUENCY FOR ASIC IMPLEMENTATIONS.

	Radix-2	Radix-4	Radix-8	Radix-64
Area	265,57	345,82	585,62	1,056,18
Delay	3.17	6.16	11.86	19.89
Frequency	200 Mhz	135 Mhz	83 Mhz	50 Mhz

The increased area of the radix-4 multiplier with respect to the radix-2 is almost 30%. The increase in the radix-8 with respect to the radix-4 is 69%, and the increase in the radix-64 with respect to the radix-8 is 80%. The growth in area is even with the radix, except in radix-8, where the reduction in the number of operands in CSA is less efficient.

The critical path delay increases with the radix. The delay of the radix-4 multiplier increases by 94% with respect to the radix-2 multiplier. The radix-8 delay increases by 92.5% with respect to the radix-4, and the radix-64 delay increases by 68% with respect to the radix-8. However, the minimum latency is reduced in same proportion. For the radix-2 implementation the latency is 3246 ns, for the radix-4 it is 3153 ns, for the radix-8 it is 3036 ns and for radix-64 it is 637 ns. It can be seen that the radix-64 multiplier implementation is the most suitable for high speed applications, while for area critical implementations the radix-2 is the most suitable, because the latency is very similar but has the smaller area.

V. Conclusions

In this paper the implementation of various structures of modular multiplier for the Montgomery algorithm has been presented. These structures have different radix (radix-2, radix-4, radix-8 and radix-64) and all are implemented with carry save adders to enhance performance. The radix-2, radix-4 and radix-8 structures maintain input and output data and intermediate results in redundant format. Radix-64 implementations only maintain intermediate results in redundant format. In high-radix implementations, the calculation of the multiples of the multiplicand and the modules requiring arithmetic operations are performed using the CSA adders structure, and do not need to be pre-calculated and stored in memory. The use of the Booth encoding scheme has helped to reduce the amount of partial products, which cannot be achieved by a shifting process.

Each of the structures has been described in VHDL, and implemented with FPGA (Xilinx) and ASIC (CMOS 130 nm) technology. The results show that the increase in the radix means an increase in area and a decrease in the maximum operation frequency. The reduction in the number of clock cycles in high-radix implementations offsets these disadvantages. The radix-64 implementation is therefore the fastest, being the best choice for applications where operating speed is critical. For applications requiring a low consumption of resources the radix-2 implementation is the one that offers better performance.

References

[1] W. Diffie and M.F Hellman "New Direction in Cryptology" *IEEE on Information Theory* 1976, vol22, no.6, pages:644~654.

[2] Rivest R L, SHamir, L. "A Method for Obtaining Digital Signatures and Public Key Cryptosystem", *Communication ACM*, 1978.

[3] N. Koblitz "Elliptic Curve Cryptosystem" *Mathematics of computations*, 1987, 48:203-209.

[4] V. S. Miller, "Use of elliptic curves in cryptography", *in CRYPTO'85*, 1986, pp. 417-426.

[5] J-h. Zhang, X. Ting-Gang, X-y Fang, "Hardware Implementation of Improved Montgomery's Modular Multiplication Algorithm", *Proc. of 2009 Int. Conf. on Communications and Mobile Computing*, pp. 370-374.

[6] P. L. Montgomery. "Modular Multiplication without trial division", *Mathematics of Computation*, April 1985, 44(170):519-521.

[7] S.S. Ghoreishi, M.A. Pourmina, H. Bozorgi, M. Dousti, "High Speed RSA Implementation Base don Modified Booth´s Technique and Montgomery´s multiplication for FPGA Platform",*Second International conference on Advances in Circuits, Electronics and Micro-electronics, 2009, pp. 86-93.*

[8] "High-Throughput FPGA Implemetation of 256-bit Montgomery Modular Multiplier", International workshop on Education Technology and Computer Science (ETCS), 2010, pp 173:176.

[9] C. K. Koc, T,Acar, and B. S. Kaliski, Jr, " Analyzing and comparing Montgomery multiplication algorithms", *IEEE micro*, 1996, 16(3):26-33.

[10] D.E.Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2, Reading, MA: Addison-Wesley, Second edition, 1981.

[11] N. Nedjah and L. de M. Mourelle, "Three Hardware Implementation for Binary Modular Exponentiation Sequential, Parallel and Systolic", *Computer Architecture and High Performance Computing*,2003.

[12] A. A. Tiountchik , "Systolic modular exponentiation via Montgomery algorithm", *Electronics Letters*,1998, Volume: 34 , Issue: 9.

[13] J.B. Shin, J. Kim; H. Lee-Kwang; "Optimization of Montgomery modular multiplication algorithm for systolic arrays", *Electronics Letters*, 1998, Volume: 34, Issue: 19.

[14] A. Tenca and C. Koc, "A Scalable Architecture for Modular Multiplication Based on Montgomery´s Algorithm", *IEEE Trans. Computers*, 2003, vol. 52, no. 9:1215-1221.

[15] D. Harris, R. Krishnamurthy and M. Anders, "An Improved Unified Scalable Radix-2 Montgomery Multiplier", *Proceedings 17th IEEE Symposium on Computer Arithmetic*, 2005.

[16] Z. Chen, Y. Sun and G. Bai, "A scalable architecture of high-performance Montgomery multiplier for design reuse",. *Proceedings. 5th International Conference on ASIC*, 2003.

[17] C. McIvor, M. McLoone and J.V. McCanny, "Modified Montgomery modular multiplication and RSA exponentiation techniques", *IEEE Proc.-Comput. Digi*t. Tech. , November 2004, Vol. 151, No. 6.

[18] J. Leu, , and A. Wu, "Design methodology for Booth-encoded Montgomery module design for RSA cryptosystem", *Proc. IEEE Int. Symp. Circuits and Systems* (ISCAS-2000) , May 2000, pp. 357-360.

[19] T. W. Kwon, C.S. You, W.S. Heo, Y.K. Kang, and J. R. Choi, "Two implementation methods of a 1024-bit RSA Cryptoprocessor based on modified Montgomery algorithm", *in Proc. IEEE Int, Symp. Circuits Syst.*, May 2001, vol. 4, pp. 650-653.

[20] Y. Fan, X. Zeng, Y. Y. Wang, H. Deng, Q. Zhang , "High Speed Radix-16 Design of a Scalable Montgomery Multiplier", *6th International Conference On ASIC, ASICON*, 2005.

[21] T. Blum and C. Paar , "High-Radix Montgomery Modular Exponentiation on Reconfigurable Hardware", *IEEE transactions on computers*, July 2001, vol. 50, no. 7.

[22] G. Sassaw, CJ. Jiménez, M. Valencia, "Influencia de la caracterización en el flujo de diseño de circuitos CMOS nanometricos", 2010.