

A Complexity-Effective Version of Montgomery's Algorithm

Viktor Bunimov, Manfred Schimmler, Boris Tolg

Institute for Computer Engineering and Communication Networks, Technical University of Braunschweig, Germany, E-mail: v.bunimov@tu-bs.de, m.schimmler@tu-bs.de, b.tolg@tu-bs.de

Abstract--A new version of Montgomery's algorithm for modular multiplication of large integers and its implementation in hardware is presented. It has been designed to meet the predominant requirements of most modern devices: small chip area and low power consumption. The algorithm is superior to the original method by a factor of 2, with respect to both area and latency. The new method has a simple structure. It requires a small amount of precomputation and storage in order to reduce the number of necessary additions by a factor of 2.

Index terms—modulo multiplication, carry save addition, Montgomery algorithm

A. INTRODUCTION

Modular Multiplication is a central operation in many application areas including public key cryptography. Given a wordlength of n bits, an n -bit integer M called the modulus, and two n -bit operands X and Y the problem is the computation of $X \times Y \bmod M$. Most devices in which this problem is essential have strict limitations in chip area and power consumption, while the typical problem size n is rather large (e.g. $n=1024$). Therefore, there is a need for algorithms that are on the one hand fast and on the other hand area and power efficient if implemented in hardware. A good complexity measure for these algorithms is the product of area and time, AT. There is a variety of solutions with an AT complexity of $O(n^2)$. But not only the asymptotic complexity is interesting for the application but also the constant factor. This is the reason why in almost all applications of modular multiplication the algorithm of Montgomery [11] is chosen. It has an AT complexity of $8n^2$ which is asymptotically a factor $\log n$ better than the well known interleaved modular multiplication [20, 21, 22] $6n^2 \log n$. In comparison to the school method [12, 16], which is simple, but unsuitable for area or power sensitive environments it is better by a factor of $3/2 \log n$. Other efficient algorithms with time complexity $\log n$ [17] must pay for their speed with an area requirement of $O(n^2)$ resulting in $AT = O(n^2 \log n)$. In this paper we present a new version of the algorithm of Montgomery that has an AT complexity of $2n^2$ and thus is superior to Montgomery's original method again by a factor of 4.

Many modifications of the above algorithms have been published by different researchers in the past that speed them up by some quantities. Brickel developed a fast implementation of the school method [6, 19] with an AT complexity of $O(n^2 \log n)$. A slight improvement of the interleaved method with the same asymptotic complexity is discussed in [20, 21, 22]. Koc found a tricky way to speed up the interleaved modular multiplication by a factor of $\log n$ [2, 18, 19]. He uses three carry save adders in the loop of the algorithm and a fast method to compare redundant numbers to 0. His version of the interleaved method results in an AT complexity of $6n^2$. All these algorithms can be improved by a constant factor by using a higher radix than 2 [9, 23, 24]. We do not explain these techniques here in detail because they can be applied with the same performance gain in all the algorithms discussed here, including our new one.

This paper is organized as follows. The standard Montgomery method for modular multiplication is given in Section B. In Section C we present the new algorithm and an improved version. The complexity analysis and the comparison to the methods previously known is summarized in section D. Finally, Section E gives some outlook to future work and concludes the paper.

B. MONTGOMERY ALGORITHM

The modular multiplication problem is defined as the computation of $P = X \times Y \bmod M$ given the integers X, Y, M with $0 \leq X, Y < M$. For practical applications only the case of odd M is interesting, where $M < 2^n \leq 2M$, i.e. the most significant bit $n-1$ of M is 1, too. There have been several approaches for computing P in hardware. Residue number systems, for example, have been recently receiving more attention [4]. Reported implementations, however, have ignored the need for conversion between binary and residues numbers. This has discouraged researchers from using such an approach [14]. Another approach is based on look-up tables, i.e. ROM-based methods. This method is quite expensive since the required memory space grows exponentially with the word size [14].

The classical method to compute modulo multiplication is by performing the multiplication and then subtracting the modulus several times until the result is less than the modulus. This approach is inefficient and suffers from low speed. The idea of Montgomery is to reduce the lengths of the intermediate results to a fixed quantity of $n+1$ bits. This is achieved by interleaving the computations and additions of new partial products with divisions by 2, each of them reducing the bit-length of the intermediate result by one.

Algorithm 1: Montgomery multiplication

Inputs: X, Y, M with $0 \leq X, Y < M$

Output: $P = (X \times Y \times (2^n)^{-1}) \bmod M$

n : number of bits in X ,

x_i : i^{th} bit of X

p_0 : LSB of P

1. $P := 0$;
2. **for** $i:=0$ **to** $k-1$ **do**
3. $P := P + x_i * Y$;
4. $P := P + p_0 * M$;
5. $P := P \text{ div } 2$
6. **if** $P \geq M$ **then** $P := P - M$;

Most existing modular multiplication solutions are based on Montgomery's algorithm [1, 5, 7, 12]. The basic idea of Montgomery is the following: adding a multiple of M to the intermediate results does not change the value of the final result, because the result is computed modulo M . M is an odd number. So, after each addition in the inner loop the LSB of the intermediate result is inspected. If it is 1, i.e. the intermediate result is odd, we add M to make it even. This even number can be divided by 2 without remainder. This division by 2 reduces the intermediate result to $n+1$ bits again. After n steps these divisions add up to one division by 2^n . Montgomery Multiplication requires two passes through the same multiplication process, therefore doubling the computation time. The first pass computes $P = (X * Y * (2^n)^{-1}) \bmod M$ and the second pass computes $(P * 2^{2n} * (2^n)^{-1}) \bmod M = (X * Y) \bmod M$ which is the desired result. On the other hand, it is very easy to implement since it operates least significant bit first and does not require any comparisons. Therefore, it can be implemented using carry save adders and a redundant representation of the intermediate results [18]. Carry save adders have a latency of $O(1)$ in comparison to standard ripple carry adders with a latency of $O(n)$ and carry lookahead adders with a latency of $O(\log n)$. Their disadvantage is that they add three operands to two results, thus producing a result in redundant form which does not allow comparisons to other values in constant time. A modification of Montgomery's Algorithm with carry save adders is given in algorithm 2:

Algorithm 2: Fast Montgomery multiplication

Inputs: X, Y, M with $0 \leq X, Y < M$

Output: $P = (X \times Y \times (2^n)^{-1}) \bmod M$

n : number of bits in X ,

x_i : i^{th} bit of X

s_0 : LSB of S

1. $S := 0; C := 0;$
2. **for** $i:=0$ **to** $k-1$ **do**
3. $S, C := S + C + x_i * Y;$
4. $S, C := S + C + s_0 * M;$
5. $S := S \text{ div } 2; C := C \text{ div } 2;$
6. $P := S + C$
7. **if** $P \geq M$ **then** $P := P - M;$

In this algorithm the delay of one pass through the loop is reduced from $O(\log n)$ to $O(1)$. We take the notion of an assignment of the sum of three operands to two values S, C as in order to indicate the use of a carry save adder.

Of course, the additions in step 6 and 7 are conventional additions. But since they are performed only once while the additions in the loop are performed n times this is subdominant with respect to the time complexity.

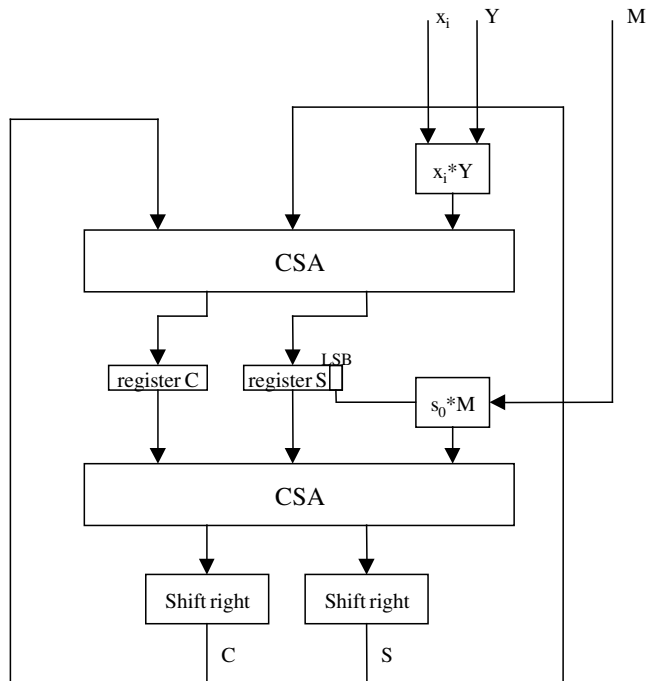


Figure 1: Implementation of loop of algorithm 2

Figure 1 shows a floor plan for the hardware implementation of algorithm 2. The dominant parts both for area and time are the two carry save adders required to compute the values of S and C in the loop of Algorithm 2. Each carry save adder consists of $n+1$ full adders.

C. THE NEW MODULAR MULTIPLICATION SCHEME

The amount of hardware for the implementation of the loop of algorithm 2 can be significantly reduced if we precompute the four possible values to be added to the intermediate result within the loop. There are four cases:

(i) if the sum of the old values of S and C is an even number, and if the actual bit x_i of X is 0, then we don't need to increment S and C at all, i.e. we add 0 before we perform the reduction of S and C by division by 2.

(ii) if the sum of the old values of S and C is an odd number, and if the actual bit x_i of X is 0, then we must add M to make the intermediate result even. Afterwards, we perform the reduction of S and C by division by 2.

(iii) if the sum of the old values of S and C is an even number, and if the actual bit x_i of X is 1, but the increment $x_i * Y$ is even, too, then we don't need to add M to make the intermediate result even. Thus, in the loop we add Y before we perform the reduction of S and C by division by 2. The same action is necessary if the sum of S and C is odd, and if the actual bit x_i of X is 1, and Y is odd as well. In this case, $S+C+Y$ is an even number, too.

(iv) if the sum of the old values of S and C is odd, the actual bit x_i of X is 1, but the increment $x_i * Y$ is even, then we must add Y and M to make the intermediate result even. Thus, in the loop we add $Y+M$ before we perform the reduction of S and C by division by 2. The same action is necessary if the sum of S and C is even, and the actual bit x_i of X is 1, and Y is odd. In this case, $S+C+Y+M$ is an even number, too.

The computation of $Y+M$ can be done prior to going through the loop. This saves one of the two additions which is replaced by the choice of the right operand to be added to the old values of S and C . Algorithm 3. is a modification of Montgomery's method which takes advantage of this idea:

Algorithm 3: Faster Montgomery multiplication

Inputs: X, Y, M with $0 \leq X, Y < M$

Output: $P = (X \times Y \times (2^n)^{-1}) \bmod M$

n : number of bits in X ,

x_i : i^{th} bit of X

s_0 : LSB of S , c_0 : LSB of C , y_0 : LSB of Y

R : precomputed value of $Y+M$

1. $S := 0; C := 0;$
2. **for** $i:=0$ **to** $k-1$ **do**
3. **if** $(s_0 = c_0)$ and not x_0 **then** $I := 0;$
4. **if** $(s_0 \neq c_0)$ and not x_0 **then** $I := M;$
5. **if** not $(s_0 \oplus c_0 \oplus y_0)$ and x_0 **then** $I := Y;$
6. **if** $(s_0 \oplus c_0 \oplus y_0)$ and x_0 **then** $I := R;$
7. $S, C := S + C + I;$
8. $S := S \text{ div } 2; C := C \text{ div } 2;$
9. $P := S + C$
10. **if** $P \geq M$ **then** $P := P - M;$

The operation \oplus in the above algorithm is the “exclusive or operation” (XOR). The advantage of algorithm 3 in comparison to algorithm 2 can be seen in the implementation of the loop of algorithm 3 in Figure 2. The possible values of I are stored in a lookup-table which is addressed by the actual values of x_i , y_0 , s_0 , and c_0 . The operations in the loop are now reduced to one table lookup and one carry save addition. Both these activities can be performed concurrently. Note that the shift right operations that implement the division by 2 can be done by routing.

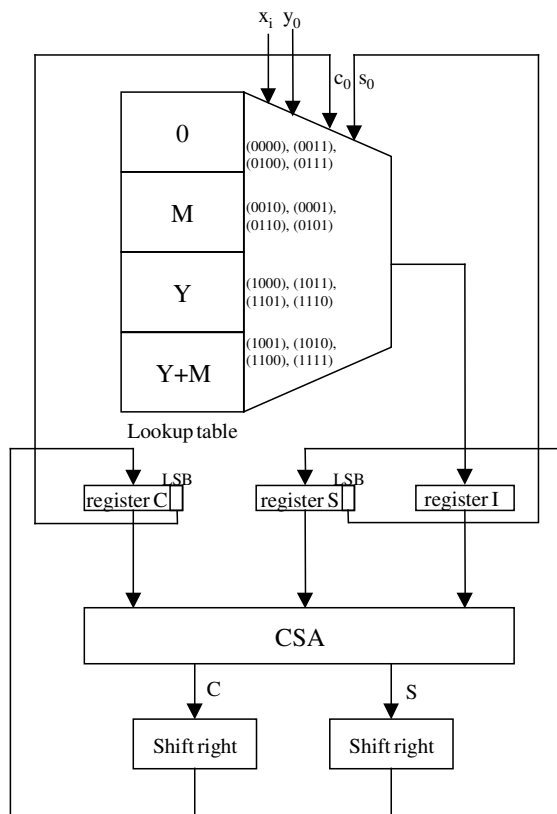


Figure 2: Implementation of loop of algorithm 3

PROPOSITION -- After execution of Algorithm 3 holds $P = X \times Y \bmod N$.

PROOF – It is sufficient to show that the i -th pass of the loop of algorithm 3 computes the same values S and C as the corresponding pass i in algorithm 2 does. This is obviously the case for $i < 0$. We discuss the possible cases for x_i , y_0 , s_0 , and c_0 for $i \geq 0$:

If $x_i = 0$ and S and C are both even, then algorithm 2 leaves S and C unchanged until they are divided by 2. The same holds if S and C are both odd, because the first carry save adder sets s_0 to 0. Exactly the same is done in algorithm 3 by assigning 0 to I in step 3.

If $x_i = 0$ and exactly one of S and C is even and the other one is odd, then algorithm 2 adds M to make the intermediate result even. We achieve this by assigning M to I in step 4 of algorithm 3.

Let now $x_i = 1$. In this case algorithm 2 adds Y to S and C in step 3. The result is an even number, exactly if y_0 , s_0 , and c_0 have been of even parity before this addition. Then algorithm 2 does not change the result in step 4, since the new value of s_0 will be 0. The corresponding operation is done in algorithm 3. I is set to be Y in step 5 and so S , C , and Y are added in step 7, producing the same values for S and C as algorithm 2.

If the result after step 3 of algorithm 2 is odd, the values of y_0 , s_0 , and c_0 must have been of odd parity before this addition. In this case the new value of s_0 is 1, and M is added to S and C in step 4. Together S and C are incremented by Y and M in this pass of the loop. The same is done in algorithm 3: $R = Y+M$ is assigned to I in step 6, and so step 7 generates the sum of S , C , Y and M , too.

D. PERFORMANCE EVALUATION

The area requirements of algorithms 2 and 3 can be seen in figures 1 and 2: The predominant area consumers are the adders. Registers for the intermediate results S and C are required in either implementation. We have added one register for I in algorithm 3 for simplicity. It could be avoided by some minor changes. The registers for Y and M in Algorithm 2 are replaced by a lookup-table in algorithm 3 which is of comparable area, if one takes into account that no units for the computation of x_i*Y and s_0*M are necessary. So, if we neglect the area for the registers the new algorithm reduces the area requirements by a factor of two.

The time performance is also reduced by a factor of two. In the loop of algorithm 2 the signals have to pass two adders, one after the other, because s_0 must be interpreted to generate one of the operands for the second addition. In algorithm 3 only one addition is required and the generation of the subsequent operands can be done concurrently.

Together we get a factor of four in the product AT of area and time.

E. OUTLOOK AND CONCLUSIONS

In this paper we have given a new version of Montgomery's algorithm for modular multiplication of large integers as used in the security units of mobile devices. It has been shown that it is well suited for an complexity efficient implementation with respect to both area and time. Algorithm 3 is an improvement by one adder unit in comparison with the standard Montgomery modular multiplication method which leads to a time reduction of a factor of two.

This new technique can be further exploited to gain efficiency in many applications. Exponentiation for RSA cryptography can be done with double speed, standard multiplication and division can be accelerated. It will be interesting to investigate how much this technique can be further improved by use of higher radix systems to reduce the number of passes through the inner loop of the algorithms.

F. REFERENCES

- [1] Walter, Colin D.: Precise Bounds for Montgomery Modular Multiplikation and Some Potentially Insecure RSA Moduli, CT-RSA 2002: San Jose, CA, USA <http://link.springer.de/link/service/series/0558/bibs/2271/22710030.htm>
- [2] Kim, Y. S., Kang, W. S., Choi, J. R.: Implementation of 1024-bit modular processor for RSA cryptosystem, School of Electronic and Electrical Engineering, Kyungpook National University, 1370 Sankyok-Dong, Book-Gu, Taegu, Korea, 702-701, www.ap-asic.org/2000/proceedings/10-4.pdf
- [3] Adi, W.: *Fuzzy Modular Arithmetic for Cryptographic Schemes with Applications to Mobile Security*, Proc. IEEE International Conference EuroComm 2000, pp. 263-265, IEEE 2000.
- [4] Bajard, J., Didier, L., Kornerup, P.: *An RNS Montgomery modular multiplication algorithm*, IEEE Transactions on Computers, Vol. 47, pp. 766-776, July 1998.
- [5] Blum, T., Paar, C.: *Montgomery Modular Multiplication on Reconfigurable Hardware*, 14th IEEE Symposium on Computer Arithmetic (ARITH-14), April 14-16, IEEE 1999.
- [6] Brickell, E.F.: *A Fast Modular Multiplication Algorithm with Application to Two Key Cryptography*, Proc. of Crypto'82, pp. 51-60, Plenum Press 1983.

- [7] Eldridge, S.E., Walter, C.D.: *Hardware implementation of Montgomery's modular multiplication algorithm*, IEEE Transaction on Computers, Vol. 42, pp. 693-699, July 1993.
- [8] Koc, C.K., Acar, T., Kaliski, B.S.: *Analyzing and Comparing Montgomery Multiplication Algorithms*, IEEE Micro, pp. 26-33, June 1996.
- [9] Blum, T, Paar, C.: High Radix Montgomery Modular Exponentiation on Reconfigurable Hardware, IEEE Transactions on Computers, July 2001 (Vol. 50, No. 7).
- [10] Menezes et. al: *Handbook of Applied Cryptography*, CRC Press, 1997.
- [11] Montgomery, P. L.: *Modular Multiplication without trial division*, Mathematics of Computation, 44, pp. 519-521, 1985.
- [12] Shand, M., Vuillemin, J.: *Fast implementation of RSA cryptography*, Proc. 11th IEEE Symposium on Computer Arithmetic, pp. 252-259, IEEE 1993.
- [13] Stinson D. R.: *Cryptography, Theory and Practice*, CRC Press, 1995.
- [14] Wu, C., Chou, Y.: *General Modular Multiplication by Block Multiplication and Table Lookup*, IEEE Int. Symposium on Circuits and System, ISCAS'94, pp. 295-298, IEEE 1994.
- [15] Yong-Yin, J., Burleson, W.: *VLSI array algorithms and architectures for RSA modular multiplication*, IEEE Transactions on VLSI Systems, Vol. 5, pp. 211-217, June 1997.
- [16] Knuth, D. E. the Art of Computer Programming, Vol. 2, Seminumerical Algorithms. Reading, MA: Addison-Wesley, Nov. 1971, 2nd printong, p. 423.
- [17] Walter, Kolin D.: Logarithmic Speed Modular Multiplication. Electronics Letters 30 (1994), No 17. pp. 1397-8.
- [18] Koc, C. K.: RSA Hardware Implementation, RSA Laboratories, RSA Data Security, Inc. August 1995, <http://security.ece.orst.edu/koc/papers/reports.html>
- [19] Jüri Poldre: Cryptoprocessor PLD001, Master Thesis, Department of Computer science, Tallinn Technical University, June 1998, www.pld.ttu.ee/~prj/master.pdf
- [20] Bertil Schmidt, Manfred Schimmler and Wael Adi: Area Efficient Modular Arithmetic for Mobile Security, ICWN'02, Las Vegas, USA (2002).
- [21] Blakley, G. R.: A Computer Algorithm for Calculating the Product $A*B$ modulo M , IEEE on Computers, Vol. c-32, No. 5, May 1983, pp. 497-500.
- [22] Sloan, K. R.: Comments on "A Computer Algorithm for Calculating the Product $A*B$ modulo M ", IEEE Transactions on Computers, Vol. c-34, No. 3, March 1985.
- [23] Morita, H.: A Fast Modular-multiplikation Algotithm based on a Higher Radix, NTT Communications and Information Processing Laboratories 3-9-11, Midori-cho, Musashino-shi, Tokyo, 180 Japan, <http://link.springer.de/link/service/series/0558/bibs/0435/04350387.htm>
- [24] Takago, Naomi: A Radix-4 Modular Multiplication Hardware Algorithm for Modular Exponentiation, IEEE Transactions on Computers, Vol. 41, No. 8, August 1992