Yet Another Hardware Implementation of Modular Multiplication

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

Abstract—

Modular multiplication is fundamental to several public-key cryptography systems such as the RSA encryption system. It is also the most dominant part of the computation performed in such systems. The operation is time consuming for large operands. This paper examines the characteristics of yet another architecture to implement modular multiplication. An experimental modular multiplier prototype is described and some simulation results are presented.

Key Words—**Modular Multiplication, Cryptosystems**

I. Introduction

The modular exponentiation is a common operation for scrambling and is used by several public-key cryptosystems, such as the RSA encryption scheme [RIV 78]. 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 are widely studied as in [BRI 89], [WAL 93], [ELD 93].

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 modular multiplier and explain in details how it executes a single multiplication. Then we comment the simulation results obtained for such an architecture.

II. Multiplication Algorithm

Algorithms that formalise the operation of multiplication generally consist of two steps: one generates a partial product and the other accumulates it with the previous partial products. The most basic algorithm for multiplication is based on the add-and-shift method: the shift operation generates the partial products while the add step sums them up [RAB 95].

The straightforward way to implement a multiplication is based on an iterative adder-accumulator for the generated partial products as depicted in Figure 1. However, this solution is quite slow as the final result is only available after n clock cycles, n is the size of the operands.

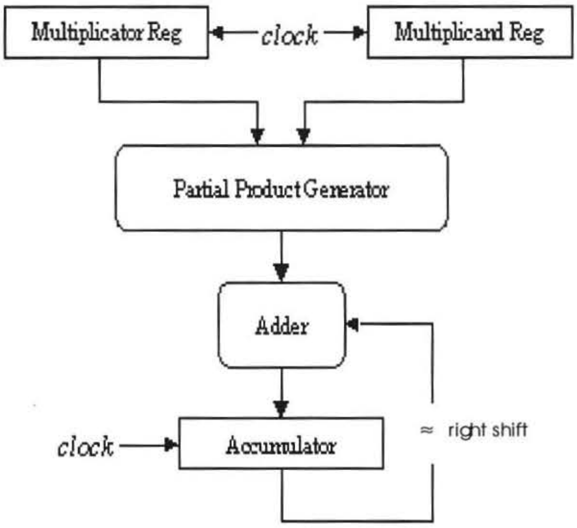

Fig. 1: Iterative multiplier.

[*] This author is a lecturer at University Veiga de Almeida, Rio de Janeiro, Brazil.

A faster version of the iterative multiplier should add several partial product 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. In this paper, we use such a parallel multiplier as described in Figure 2. Now, we detail the algorithms used to compute the partial products and to sum them up.

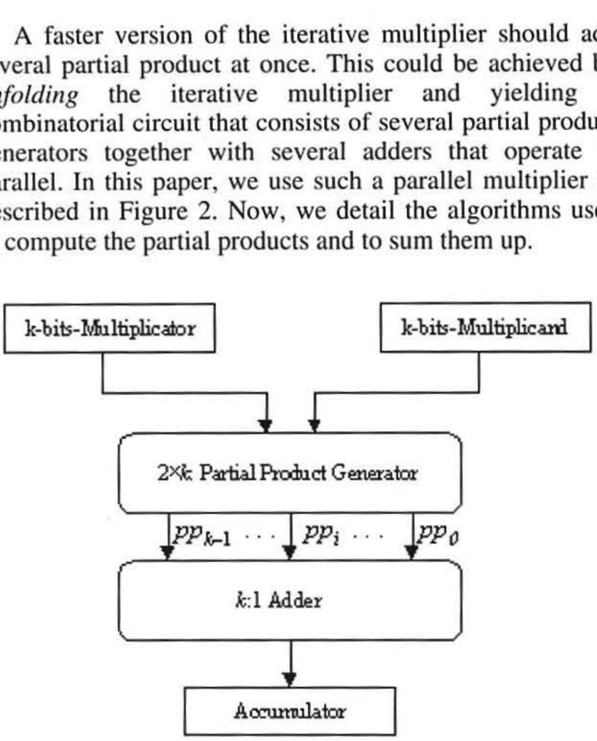

Fig. 2: Parallel multiplier.

Now, we concentrate on the algorithm used to compute partial products as well as reducing the corresponding number without deteriorating the space and time requirement of the multiplier.

Let X and Y be the multiplicand and multiplicator respectively and let n and m be their respective sizes. So, we denote X and Y as follows:

$$X = \sum_{i=0}^{n} x_i \times 2^i \text{ and } Y = \sum_{i=0}^{m} y_i \times 2^i$$

$$\Rightarrow X \times Y = \sum_{i=0}^{n} x_i \times Y \times 2^i$$

Inspired by the above notation of X, Y and that of $X \times Y$, the add-and-shift method [RAB 95] generates n partial products: $x_i \times Y$, $0 \leq i < n$. Each partial product obtained is shifted left or right depending on whether the starting bit was the less or the most significant and added up. The number of partial products generated is bound above by the size (i.e. number of bits) of the multiplier operand. In cryptosystems, operands are quite large as they represent blocks of text (i.e. ≥ 1024 bits).

Another notation of X and Y allows to halve the number of partial products without much increase in space requirements. Consider the following notation of X and $X \times Y$

$$X = \sum_{i=0}^{\lceil (n+1)/2 \rceil - 1} \tilde{x}_i \times 2^{2\times i}, \quad \text{where} \quad \tilde{x}_i = x_{2\times i-1} + x_{2\times i} - 2 \times x_{2\times i+1}$$

$$\text{and} \quad \tilde{x}_{-1} = \tilde{x}_n = \tilde{x}_{n+1} = 0$$

$$X \times Y = \sum_{i=0}^{\lceil (n+1)/2 \rceil - 1} \tilde{x}_i \times Y \times 2^{2\times i}$$

The possible values of $\tilde{x}_i$ with the respective values of $x_{2\times_i+1}$, $x_{2\times_i}$, and $x_{2\times_i-1}$ are –2 (100), –1 (101, 110), 0 (000, 111), 1 (001, 010) and 2(011). Using this recoding will generate $\lceil (n+1)/2 \rceil - 1$ partial products.

Inspired by the above notation, the modified Booth algorithm [BOO 86], [MAC 61], [BEW 94] generates the partial products $\tilde{x}_i \times Y$. These partial products can be computed very efficiently due to the digits of the new representation $\tilde{x}_i$. The hardware implementation will be detailed in Section 3.

In the algorithm of Figure 3, the terms $4 \times 2^{n+1}$ and $3 \times 2^{n+1}$ are supplied to avoid working with negative numbers. The sum of these additional terms is congruent to zero modulo $2^{n+\lceil (n+1) \rceil - 1}$. So, once the sum of the partial products is obtained, the rest of this sum in the division by   is finally the result of the multiplication $X \times Y$.

```
Algorithm Multiplier($x_{2\times i-1}$, $x_{2\times i}$, $x_{2\times i+1}$, Y) {
    int product =0;
    int pp[ $\lceil (n+1)/2 \rceil - 1$ ];
    pp[0]= ( $\tilde{x}_0 \times Y + 4 \times 2^{n+1}) \times 2^{2\times i}$ ;
    for i=0 to    {
        pp[i] = ( $\tilde{x}_i \times Y + 3 \times 2^{n+1}) \times 2^{2\times i}$ ;
        product = product + pp[i];
    }
    return product mod $2^{n+\lceil (n+1)/2 \rceil - 1}$ ;
}
```

Fig. 3: Multiplication algorithm.

III. Reduction Algorithm

A modular reduction is simply the computation of the remainder of an integer division. It can be denoted by:

$$X \bmod M = X - \left\lfloor \frac{X}{M} \right\rfloor \times M$$

However, a division is very expensive even compared with a multiplication. Using Barrett's method [BAR 86], [SHI 97], we can estimate the remainder using two simple multiplications. The approximation of the quotient is calculated as follows:

$$\left\lfloor \frac{X}{M} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{X}{2^{n-1}} \right\rfloor \times \left\lfloor \frac{2^{n-1} \times 2^{n+1}}{M} \right\rfloor}{2^{n+1}} \right\rfloor \cong \left\lfloor \frac{\left\lfloor \frac{X}{2^{n-1}} \right\rfloor \times \left\lfloor \frac{2^{2\times n}}{M} \right\rfloor}{2^{n+1}} \right\rfloor$$

The equation above can be calculated very efficiently as division by a power of two 2^x are simply a truncation of the operand' x-least significant digits. The term $\lfloor 2^{2\times n}/M \rfloor$ depends only on the modulus M and is constant for a given modulus, hence, can be pre-computed and saved in an extra register. Hence the approximation of the remainder using Barrett's method [BAR 86], [SHI 97] is a positive integer smaller than 2×(M−1). So, one or two subtractions of M might be required to yield the exact remainder.

IV. Modular multiplier architecture

In this section, we outline the architecture of the multiplier, which is depicted in Figure 4. Later on in this section and for each of the main parts of this architecture, we give the detailed circuitry, i.e. that of the *partial product generator*, *adder* and *reducer*.

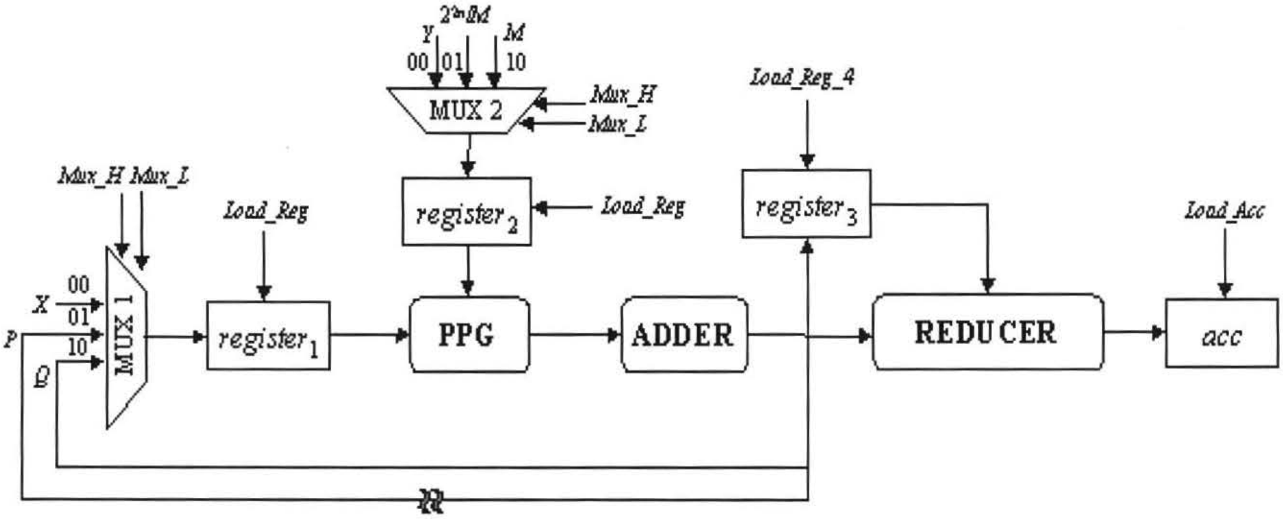

Fig. 4: The modular multiplier architecture.

The multiplier of Figure 4 performs the modular multiplication $X \times Y$ mod M in three main steps:

1. Computing the product $P = X \times Y$;
2. Computing the estimate quotient $Q = P/M$ $\Rightarrow Q \cong P/2^{n-1} \times \lfloor 2^{2\times n}/M \rfloor$;
3. Computing the final result $P - Q \times M$.

During the first step, the modular multiplier first loads *register*$_1$ and *register*$_2$ with X and Y respectively; then waits for *PPG* to yield the partial products and finally waits for the *ADDER* to sum all of them. During the second step, the modular multiplier loads *register*$_1$, *register*$_2$ and *register*$_3$ with the obtained product P, the pre-computed constant $\lfloor 2^{2\times n}/M \rfloor$and P respectively; then waits for *PPG* to yield the partial products and finally waits for the *ADDER* to sum all of them. During the third step, the modular multiplier first loads *register*$_1$ and *register*$_2$ with the obtained product Q and the modulus M respectively; then awaits for *PPG* to generate the partial products, then waits for the *ADDER* to provide the sum of these partial products and finally waits for the *REDUCER* to calculate the final result $P - Q \times M$, which is subsequently loaded in the accumulator *acc*.

A The partial product generator

The partial product generator is composed of k *Booth recoders* [BOO 86], [MAC 61], [BEW 94]. They communicate directly with k partial product generators as shown in Figure 5.

The required partial products, i.e. $\tilde{x}_i \times Y$ are *easy* multiple. They can be obtained by a simple shift. The negative multiples in 2's complement form, can be obtained form the positive corresponding number using a bit by bit complement with a 1 added at the least significant bit of the partial product. The additional terms introduced in the previous section can be included into the partial product generated as three/two/one most significant bits computed as follows, whereby, ++ is the bits *concatenation* operation, $\langle A \rangle$ is the binary notation of integer A, 0^i is a run of i zeros and $B_{[n:0]}$ is the selection of the n less significant bits of the binary representation B.

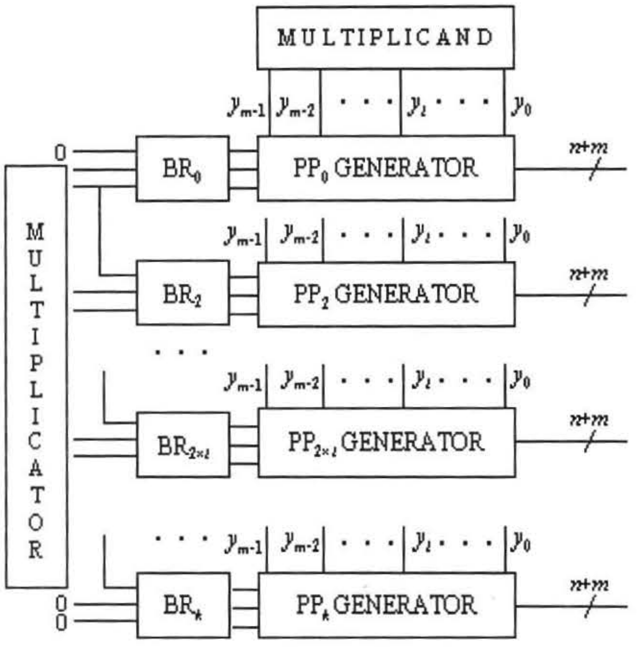

Fig. 5: The partial product generator architecture.

$$\langle pp_0 \rangle = \overline{s_0} s_0 s_0 ++ \langle \left|\tilde{x}_0\right| \times Y \oplus s_0 \rangle + s_0$$

$$\langle pp_{2\times j} \rangle = \left(1 \overline{s_{2\times j}} ++ \langle \left|\tilde{x}_{2\times j}\right| \times Y \oplus s_{2\times j} \rangle + s_{2\times j} \right) ++ 0^{2\times j}$$

for $1 \leq j < k-1$ and for $j = k-1 = k\bullet$, we have:

$$\langle pp_{2\times k'} \rangle = \left(\overline{s_{2\times k'}} ++ \langle \left|\tilde{x}_{2\times k'}\right| \times Y \oplus s_{2\times k'} \rangle + s_{2\times k'} \right) ++ 0^{2\times k'}$$

$$\langle pp_{2\times k} \rangle = \langle \left|\tilde{x}_{2\times k}\right| \times Y \rangle_{[n:0]} ++ 0^{2\times k}$$

The Booth selection logic circuitry used, denoted by BR_i for $0 \leq i \leq k$ in Figure 5, is very simple. The cell is described in Figure 6. The inputs are the three bits forming the Booth digit and outputs are three bits: the first one SY is set when the partial product to be generated is Y are $-Y$, the second one $S2Y$ is set when the partial product to be generated is $2\times Y$ are $-2\times Y$, the last bit is the simply the last bit of the Booth digit given as input. It allows us to complement the bits of the partial products when a negative multiple is needed.

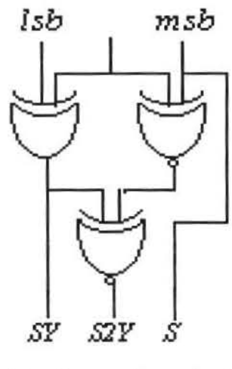

Fig. 6: The Booth recoder selection logic.

The circuitry of the partial generator denoted by *PP$_i$ Generator*, is given in Figure 7.

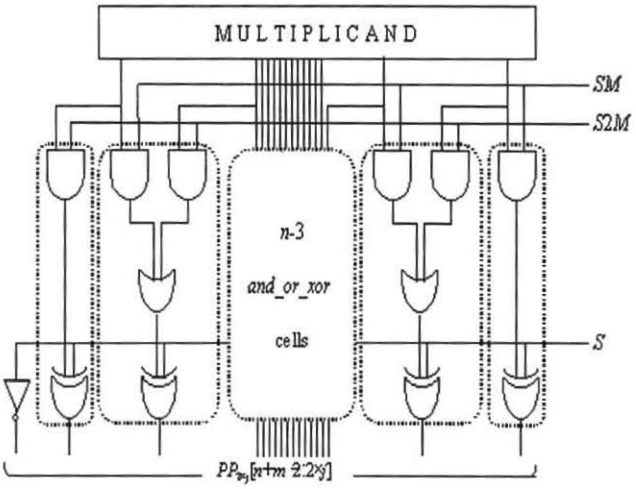

Fig. 7: The partial product generator.

B The adder

In order to implement the adder of the generated partial products, we use a hybrid new kind of adder. It consists cascade of intercalated stages of *carry save adders* and *delayed carry adders*.

The carry save adder (CSA) is simply a parallel ensemble of f full adders without any horizontal connection. Its function is to add three f-bit integers a, b and c to yield two new integers *Carry* and *Sum* such that $carry + sum = a + b + c$. The pair (*carry*, *sum*) will be called a *delayed carry integer*. The delayed carry adder (DCA) is a parallel ensemble of f half adders. Its function is to add two delayed carry integers (a_1, b_1) and (a_2, b_2) together with an integer c to produce a single integer *sum* such that $sum = a_1 + b_1 + a_2 + b_2 + c$. The main cell of the proposed adder is depicted in Figure 8, where the partial products PP_i, $0 \leq i \leq 6$ are the input operands. Using the carry save adder, the ith bit of *carry* and *sum* are defined as $sum_i = a_i \oplus b_i \oplus c_i$ and $carry_i = a_i \times b_i + a_i \times c_i + b_i \times c_i$ respectively.

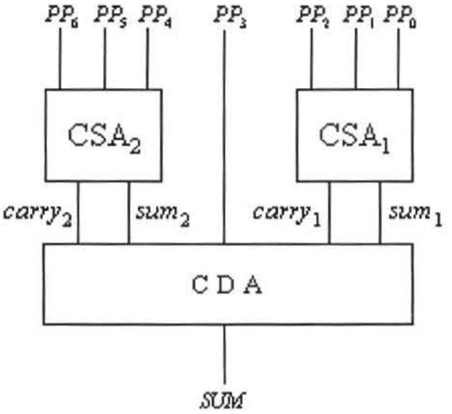

Fig. 8: The main cell of the proposed adder.

The architecture of the delayed carry adder uses 5×*n* half adders and *n* full adders as described in Figure 9. This architecture ignores the overflows. However, these can be easily estimated from the three top bits of the operands. The proof concerning the soundness of the result delivered by the adder can be found in [WAL 86].

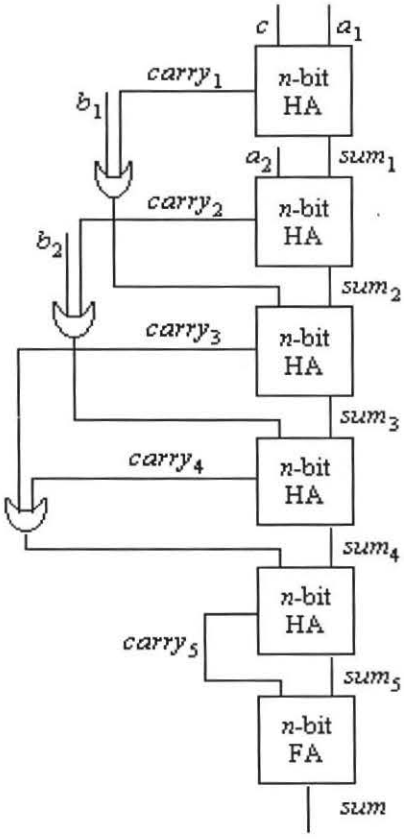

Fig. 9: The structure of the carry delayed adder.

C The reducer

The main task of the reducer consists of subtracting $Q \times M$, i.e. the product obtained in the third step of the modular multiplier from P, i.e. the product yielded in the first step of the modular multiplier. A subtraction of an p-bits integer K is equivalent to the addition of $2^p - x$. Hence the reducer simply performs the addition $P + (2^{n+m} - Q \times M)$. The latter value is simply the two's complement of $Q \times M$.

The addition is performed using a *carry look-ahead adder*. It is based on computing the carry bits C_i prior to the actual summation. The adder takes advantage of a relationship between the carry bits C_i and the input bits A_i and B_i.

$$C_i = G_{i-1} + (G_{i-2} + (G_{i-3} + \Lambda + (G_1 + (G_0 + C_0 P_0) \times P_1) \times P_2 \Lambda) \times P_{i-1})$$

whereby $G_i = A_i \times B_i$ and $P_i = A_i + B_i$. The general structure of the used carry look-ahead adder is given in Figure 10.

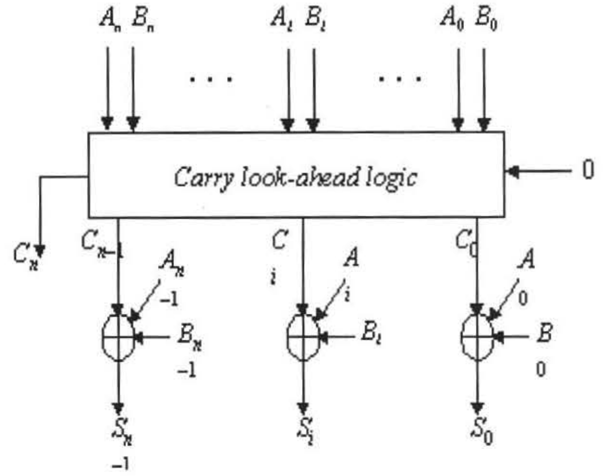

Fig. 10: The structure of the carry look-ahead adder.

V Conclusion

In this paper, an alternative architecture for computing modular multiplication based on Booth algorithm and on Barret's relaxed residiuum method is described. The Booth algorithm is used to compute the product while Barret's method is used to calculate the remainder. The architecture was validated through behavioral simulation results using the 0.6µm CMOS-AMS standard cell library. The total execution time is 3570 nanoseconds for 1024-bit operands.

One of the advantage of this modular multiplication implementation resides in the fact that it is easily scalable with respect to the multiplier and modulus lengths.

REFERENCES

[BAR 86] Barrett, P., *Implementating the Rivest, Shamir and Aldham public-key encryption algorithm on standard digital signal processor*, Proceedings of CRYPTO'86, Lecture Notes in Computer Science **263**:311-323, Springer-Verlag, 1986.

[BEW 94] Bewick, G. W., *Fast multiplication algorithms and implementation*, Ph. D. Thesis, Department of Electrical Engineering, Stanford University, United States of America, 1994.

[BOO 51] Booth, A., *A signed binary multiplication technique*, Quarterly Journal of Mechanics and Applied Mathematics, pp. 236-240, 1951.

[BRI 89] Brickell, E. F., *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.

[ELD 93] Eldridge, S. E. and Walter, C. D., *Hardware implementation of Montgomery's Modular Multiplication Algorithm*, IEEE Transactions on Computers, **42**(6):619-624, 1993.

[MAC 61] MacSorley, O., *High-speed arithmetic in binary computers*, Proceedings of the IRE, pp. 67-91, 1961.

[RAB 95] Rabaey, J., *Digital integrated circuits: A design perspective*, Prentice-Hall, 1995.

[RIV 78] Rivest, R., Shamir, A. and Aldham, L., *A method for obtaining digital signature and public-key cryptosystems*, Communications of the ACM, **21**:120-126, 1978.

[SHI 97] Shindler, V., *High-speed RSA hardware based on low-power piplined logic*, Ph. D. Thesis, Institut für Angewandte Informationsverarbeitung und Kommunikationstechnologie, Technishe Universität Graz, January 1997.

[WAL 93] Walter, C. D., *Systolic modular multiplication*, IEEE Transactions on Computers, **42**(3):376-378, 1993.

[WAL 86] Walter, C. D., *A verification of Brickell's fast modular multiplication algorithm*, International Journal of Computer Mathematics, **33**:153:169.