2009 Second International Conference on Advances in Circuits, Electronics and Micro-electronics

High Speed RSA Implementation Based on Modified Booth's Technique and Montgomery's Multiplication for FPGA Platform

S. S. Ghoreishi
Dept. of Electrical Engineering
Islamic Azad University (IAU)
Science and Research Branch, Tehran, Iran
salehghoreyshi@gmail.com

H. Bozorgi
Dept. of Electrical. & Electronics. Engineering,
University of Guilan
Rasht, Iran
hamidbozorg@gmail.com

M.A. Pourmina
Dept. of Electrical Engineering
Islamic Azad University (IAU)
Science and Research Branch, Tehran, Iran
pourmina@srbiau.ac.ir

M. Dousti
Dept. of Electrical Engineering
Islamic Azad University (IAU)
Science and Research Branch, Tehran, Iran
m_dousti@srbiau.ac.ir

***Abstract*— Rivest, Shamir and Adleman (RSA) encryption algorithm is one of the most widely used and popular public-key cryptosystem. The main step in this algorithm is modular exponentiation which can be done by a sequence of modular multiplication. Thus, modular multiplication is the major factor, in many cryptosystems, e.g. the RSA Two-Key system and in the proposed digital signature standard DSS. One of the most efficient algorithms of modular multiplication is the Montgomery multiplication. In this paper, modified radix-4 modular multiplication was developed based on Booth's multiplication technique. We use CSA (Carry Save Adder) to avoid carry propagation. Also a very fast algorithm was presented and used for computing the modular reduction. We proposed new hardware architecture for optimum implementation of this algorithm. According to our design, for encrypting an n-bit plaintext, we need to about 3/4n (n + 11) clock cycles. We used Xilinx VirtexII and XC4000 series FPGAs (Field Programmable Gate Array). As a result, it is shown that the processor can perform 1024-bit RSA operation in less than 15ms and 50ms at 54.6MHz and 16.1MHz on Xilinx VirtexII and XC4000 series FPGA, respectively. Finally we compared our results with the previous works. We can say that a significant improvement was achieved in terms of time and in terms of used time-area (TA) our work is good.**

Keywords- RSA Encryption; Montgomery multiplication; Montgomery exponentiation; Booth's Algorithm; CSA; FPGA

I. INTRODUCTION

With the increase in data communication and the expansion of Internet service like email, secure telephony mobile internet, e-commerce, e-banking and so on, security problem has become most important over the networks. Cryptographic systems can provide the objectives on information security: confidentiality, user authentication, data origin authentication, data integrity and non-repudiation. In contrast to symmetric key cryptosystems, public-key cryptosystems are capable of fulfilling all of these objectives. However, in order to be fast enough and feasibility practical in the applications mentioned above, public key schemes have to be implemented in hardware.

The RSA [1] is the most popular public key encryption scheme. In terms of the data encryption processing in RSA, it is the main operation to compute modular exponentiation by repeat modular multiplication. However, the large bit (e.g. 1024bit or 2048bit) modular operation makes the RSA systems slow. Also, it makes the hardware implementation difficult. The security of the RSA algorithm appears in the difficulty of the factorizing large integers. Improvements in the factorization algorithm may unintentionally require that the size of the key be continually and appropriately recommended. The flexibility to change key length or modify the embedded algorithm to respond to design flaws or changes in standards or data formats requires hardware reconfigurability. Reconfigurable hardware applies to a device that can be configured at runtime, to implement a function as hardware circuit. Commercially available reconfigurable devices include Field Programmable Gate Arrays (FPGA) and Complex Programmable Logic Devices (CPLD) [14, 15]. As mentioned before, the basic operation in RSA algorithm is modular exponentiation on large integers, and this operation requires a long computing time. Thus for hardware implementation of RSA we need to fast modular multipliers. There are different kinds of algorithm for modular multiplication which Montgomery is one of the best of them. Montgomery's algorithm replaces trial division by the modulus with a series of additions and divisions by a power of two, which is very useful for hardware implementation [2]. One of the times improvement techniques (of course in expense of hardware cost) are using high radix. In high radix algorithms, multi-bits of multiplicand are processed at one clock cycle, and thus the number of iterations can be reduced. Another way to speed up the RSA cryptosystem is to use Chinese Reminder Theorem (CRT) technique for decryption. CRT makes the RSA 4 times faster. There are two implementation styles for Montgomery multiplication algorithm. One is to use CSA to avoid carry propagation [3, 4] and the other is using the systolic- array element [5, 6]. For radix-2 multiplication, both methods are adequate for high-speed clock implementation. However, for high-radix multiplication, in systolic array implementation, it is

978-0-7695-3832-7/09 $26.00 © 2009 IEEE
DOI 10.1109/CENICS.2009.25

difficult to control multi-bits in 1-bit based pipelining structure and avoid latency problem. The rest of the paper is organized as follows. In Section II and III, the radix-2 modular multiplication algorithm (Montgomery's algorithm) is briefly reviewed, and the radix-4 modular multiplication algorithm (the extended Montgomery's algorithm) is then presented. Then in Section IV, we present a very fast algorithm based on Booth's multiplication technique and using CSA. Section V describes calculation of the Montgomery exponentiation. In Section VI, we describe an algorithm to calculate the modular reduction. Hardware architecture for optimum implementation of these algorithms has been presented in Sections VII and VIII. Section IX describes data and control path unit. In Section X, we conclude the paper with some final remarks and provide the implementation results and some comparisons with previously reported cases.

II. MONTGOMERY'S MULTIPLICATION

In RSA cryptosystem, the public encryption key is the pair of positive integers *(E, N)* and also the private decryption key is another pair of positive integers *(D, N)*. To encrypt plaintext message *M*, the entire message must first be represented as a sequence of integers between 0 and *M-1*, and then raised to the E^{th} power modulo *M*. Similarly, to decrypt the cipher text *C*, it must be raised to the D^{th} power modulo *M*. Therefore:

$$C = M^E \bmod N \qquad (1)$$

Decryption is done by calculation

$$M = C^D \bmod N \qquad (2)$$

As above shown both encryption and decryption require an algorithm for computing a modular exponentiation. Clearly, modular exponentiation is the main operation of the RSA and DSS algorithm. Modular exponentiation is reduced to a series of modular multiplications and squaring steps. The algorithm for modular multiplication described below has been proposed by P. L. Montgomery in 1985 [2]. It is a method for multiplying two integers modulo *M*, while avoiding division by *M*. The idea is to transform the integers in m-residue and compute the multiplication with these m-residues. Finally we transform back to the normal representation. Therefore, if we use multiplication instead of squaring, this can be done by a sequence of multiplications. Let *A*, *B* and *N* is our three n-bit integers where:

$$A = \sum_{i=0}^{n-1} a_i 2^i, B = \sum_{i=0}^{n-1} b_i 2^i, N = \sum_{i=0}^{n-1} n_i 2^i$$

Radix-2 Montgomery multiplication [2, 12] is shown as follows.

1. $S_0 = 0;$
2. *For* $i = 0 : n-1;$ *(3)*
3. $q_i = (S_i + a_i B) \bmod 2;$
4. $S_{i+1} = (S_i + aiB + q_i N)/2;$
5. *End*
6. *Return* S_n

Radix-2 denotes that the multiplier *A* is scanned one bit in each iteration of the loop, called computational loop or computational cycle. A generalization of the algorithm for a higher Radix is presented in the next Sections. The main advantage in this algorithm is that the division step in taking the modulus is substituted with division of powers of 2. The latter on the other hand is implemented by simple shift operations. The algorithm requires that the modulus N and the constant R ($R = 2^n \bmod N$) be relative primes. This is achieved by using odd numbers for the modulus, while the constant is always a power of 2. Thus, it is always true that $gcd(N, R) = 1$. The most time consuming part of the algorithm is the calculation of S given by the three input addition (step 4). It comes from the carry propagation of the very large operand additions and this can be avoided by using CSA for additions. The function of CSA is to add three n-bit inputs *X, Y* and *Z* to produce two outputs *C* and *S* as results such that $C + S = X + Y + Z$.

The final output of above algorithm is $S_n = A.B.R^{-1} \bmod N$ where $R = 2^n \bmod N$. Note that in this algorithm, if $S_n > N$ then we must subtract S_{n+1} by *N*. Two disadvantages have been observed for this algorithm. At first, this algorithm computes $S_n = A.B.R^{-1} \bmod N$ instead of $A.B \bmod N$. The second disadvantage of this algorithm is the additional subtraction in the final step of the algorithm. This subtraction is very time consuming operation and must be avoided.

III. EXTENTION OF MONTGOMERY'S MULTIPLICATION ALGORITHM IN HIGHER RADIX

We can improve the performance of Montgomery's multiplication by reducing the iterations of it which is equal to numbers of partial products. In order to decrease the numbers of partial products we can propose a radix-4 algorithm.

K: determines the radix of computation $Radix = 2^k$.

q_{Bj} : Determines the multiples of multiplicand B that are to be added to partial products in the j^{th} iteration of computational loop.

q_{Nj}: Determines the multiples of modules *N* that are to be added to the partial products *S* in the j^{th} iteration of computational loop.

Algorithm (1) shows the high-radix Montgomery's algorithm ($R\ 2^k\ MG\ 2^k$) [7].

Algorithm (1):
Input: *A, B, N and K*
Output: $A \times B \bmod N$

1. $a_{-1}=0,\ S=0$
2. *For* $j = 0 :\ k : n-1$

2.1 $q_{Bj} = Booth(a_{j+k-1..j-1})$

2.2 $S = S + q_{Bj} \times B$

2.3 $q_{Nj} = S_{k-1..0} \times (2^k - N^{-1}_{k-1..0}) \bmod 2^k$

2.4 $S = sign\ ext.\ (S + q_{Nj} \times N)/2^k$

3. If $S \geq N$ then $S = S - N$

Algorithm (1) was shown and proved in [7]. Now there is a question that which radix is better? To answer this question we must denote, in our implementation which character is more important. High radix has more speed but it is not suitable by hardware. It is basically tradeoff between computation time and silicon area. In [6], there are some figures which introduced used area and time of this algorithm on different radixes. As well as it was shown in figure1, we have time improvement by high radix but hardware cost will be increased linearity. Finally we can come to this result that, the best selection is radix-4, so utilized it in calculation of exponentiation and then for having more speed, we used Booth's technique.

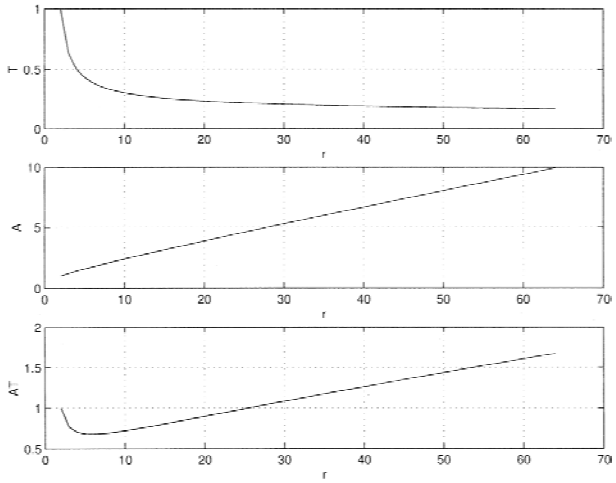

FIG.1. Area and Time Complexity of the Digit-Level Radix-r Modular Multiplier

IV. RADIX-4 MODULAR MULTIPLIER

To improve the performance of modular multiplication, numerous high-radix modular multiplication algorithms have been proposed. It can be improved by reducing the number of iterations *n* in Montgomery's algorithm. The number of iterations *n* in Montgomery's algorithm is equivalent to number of partial products. To reduce the number of partial products to be calculated, we propose a radix-4 algorithm which requires only $\lceil (n+3)/2 \rceil$ iterations. This algorithm can be implemented by Booth's multiplication technique. With Booth's Recoding scheme, the radix-4 algorithm uses the multiples of *B* and *N*, $\{\pm B, \pm 2B\}$and $\{\pm N, \pm 2N\}$, instead of $\{B, 2B, 3B\}$ and $\{N, 2N, 3N\}$ which need higher hardware cost.

Let Input: $A=B$ are two *(n+1)-bit* signed integers and *N* is an *(n+1)-bit* odd integer and $-N \leq A,B \leq N$. Output: $A.B.R^{-1}\ mod\ N$ where $R = 4^{\lceil (n+3)/2 \rceil}$.

Radix- 4 Montgomery modular multiplication algorithm [7] is shown as follows.

R4MG (A, B, N)
{
$S_0 = 0$;
For $(i = 0; i < [(n+3)/2]; i++)$ *{*
$(t_{i1}, t_{i0}) = (S_i + PP_i) \bmod 4$;
If $(t_{i0}=0)$*{/** $t_{i0} \neq n_0$ **/*
If $(t_{i1} = 0)\ S_{i+1} = (S_i + PP_i) / 4$; (4)
Else $S_{i+1} = (S_i + PP_i + 2N) / 4$ *;}*
*Else {/** $t_{i0} = n_0$ **/*
If $(t_{i1} = n_1)\ S_{i+1} = (S_i + PP_i - N) / 4$;
Else $S_{i+1} = (S_i + PP_i + N) / 4$ *;}}*
Return $S_{[(n+3/2)]}$

In above algorithm, we extend sign of *A* by two in order to guarantee that $PP_{\lceil (n+1)/2 \rceil} = 0$. Also, we see that two least significant bits (*LSB*) of $T_i = S_i + PP_i$, (t_{i1}, t_{i0}), will be used in the i^{th} iteration to determine the value of modular reduction (N_i). It is selected from $\{2N, N, -N\}$. In R4MG () Algorithm, when $t_{i1}=n_1$ and $t_{i0}=1$, $N_i = -N$ is chosen, while when $t_{i1} \neq n1$ and $t_{i0}=1$, $N_i=N$ is chosen. This guarantees that the result will be a multiple of 4. In the modified Booth's multiplier, the negative partial products are represented in two's-complement form. Since, both PP_i and N_i are *(n+2)-bit* odd integers so the sign bit of them are $PP_{i\,(n+1)}$ and $n_{i(n+1)}$ respectively. Thus the presentation of $PP_i = (PP_{i\,(n+1)} \ldots PP_{i1}\ PP_{i0})$ in two's-complements form is: $PP_i = (-pp_{i(n+1)})2^{n+1} + \sum_{j=0}^{n} pp_{ij} 2^j$). The sign bit $(PP_{i(n+1)})$ can be considered as an negative number with the weight 2^{n+1}, and in that case the *LSBs* represent a positive number that should be added by the negative number in order to calculate the accurate number of PP_i. In radix-4 Booth's multiplicand, each PP_{i+1} has two-bit offset to the left as compared with PP_i . In order to add them, it is usual to extend sign bit, so it is not proper for hardware implementation. In order to reduce hardware complexity, Roorda proposed an algorithm which doesn't need sign bit extension [8]. In this section we define the Roorda method by radix-4. The method first inverts sign bit of each partial product and then put 1 to the left side of the inversed sign bit. The reason of this work was proved in [6]. Indeed we decode the sign bit as follows: if $PP_{i(n+1)} = 0$ then $PP_{i(n+2)}PP_{i(n+1)} = 11$ and if $PP_{i(n+1)} = 1$ then $PP_{i(n+2)}\ PP_{i(n+1)} = 10$. For N_i we do the same. (We recode sign bit of N_i and B_i respectively with *sgnN* and *sgnB*). We need to one thousand twenty eight 1024-bit additions for multiplication of two 1024-bit integers by R4MG () algorithm. If we use an adder except CSA type, every addition takes several hundred *ns* long whereas the delay in CSA is equal to one flip flop (several *ns*). But, we cannot be able to define the sign bit when use CSA, and the presented algorithm need to appoint sign bit exactly. So, we proposed modified radix-4 Booth Montgomery

multiplication using CSA which is reliable for all of the above problems.

Algorithm (2) shows Montgomery's multiplication using modified radix-4 Booth's algorithm and CSA.

Algorithm(2): Inputs: $A = \sum_{i=0}^{n} a_i 2^i$

$B = \sum_{i=0}^{n} b_i 2^i$ $N = \sum_{i=0}^{n} n_i 2^i$, a_i , b_i *and* $n_i \in \{0,1\}$, $n_n = 0$

Output: $A.B.R^{-1} mod\ N$ *where* $R = 4^{[(n+3)/2]}$.

1. $C_{in}=a_{-1}=0$, $a_{n+2}=a_{n+1}=a_n$, $S=C=0$,
2. *For i=0:* $\left\lceil \frac{n+3}{2} \right\rceil - 1$ *do*
   - 2.1. $(negb, B_i) = BR((a_{2i+1}, a_{2i}, a_{2i-1}), B)$
   - 2.2. $(C, S) = CSA(S, C, B_i)$
   - 2.3. $(t_{i1}, t_{i0}) = ((S_1,S_0) + (C_1, negb) + C_{in}) \bmod 4$
   - 2.4. $negn = not\ (t_{i1} \oplus n_1)$
   - 2.5.1. *If* $(t_{i1}, t_{i0}) = 00$ *Then* $N_i=0$
   - 2.5.2. *If* $(t_{i1}, t_{i0}) = 01$ *Then* $N_i = 2N$
   - 2.5.3. *If* $(t_{i1}, t_{i0}) = 10$ *Then* $N_i = \neg N$, $C = C + negn$
   - 2.5.4. If $(t_{i1}, t_{i0}) = 11$ Then $N_i = N$
   - 2.6. $(C, S) = CSA(S, C, N_i)$
   - 2.7. $C_{in} = ((S_1,S_0) + (C_1, negb) + C_{in}) \bmod 4$
   - 2.8. $(C, S) = (C/4, S/4)$
3. $S = S+C$
4. *Return S*

In this algorithm, *(C,S) = CSA (X,Y,Z)* represent a *CSA* addition of three inputs *X,Y* and *Z*, and the two outputs of the addition are saved in *C* and *S* separately. Unlike Carry Propagation Adder (*CPA*), *CSA* does not have a separate carry-input, hence it is impossible to calculate negative multiples of 2's-complement numbers, *B* and *N* with *CSA* directly. Therefore as Table I, in order to solve this problem we should define *negb* parameter in a way that if B_i were equal to *B* or *2B* then *negb*=0 and if B_i was equal to ¬*B* or ¬*2B* then *negb=1*(¬*B* and ¬*2B* are the bit-wise inverses of *B* and *2B*). In the same manner, N_i is also represented by *negn* and like *negb*. In step 2.1 of algorithm (2), *BR* is represented of Booth decoder that selects one of the numbers on the basis of 3 successive bits of $A\ (a_{2i+1}, a_{2i}, a_{2i-1})$. Note that a 2-bit addition of two 2-bit inputs and 1-bit carry input is performed in step 2.3 and operation of mod 4 is simply done by taking 2-bit sum output as a result for the value of (t_{i1}, t_{i0}) which is used for determining N_i. The calculation of *C+negn* in step 2.5.3 can be easily done by substitution the Least Significant Bit (*LSB*) of *C*, by *negn*. Also, another 2-bit addition is used in step 2.7, and division by 4 is simply done by only taking the carry output results to update C_{in} for the next iteration. Now that the two least significant bits of the sum of the intermediate value, in step 2.8 are 0, division by 4 is just 2-bit right shift of *C* and *S*. As a result, using two 2-bit additions and carry update, we can easily implement this algorithm.

TABLE I. RADIX-4 BOOT DECODER

$a_{i+1}a_ia_{i-1}$	B_i *negb*	$a_{i+1}a_ia_{i-1}$	B_i *negb*
0 0 0	0 0	1 0 0	¬2B 1
0 0 1	B 0	1 0 1	¬B 1
0 1 0	B 0	1 1 0	¬B 1
0 1 1	2B 0	1 1 1	0 0

V. MODULAR EXPONENTIATION

As mentioned before, modular exponentiation can be done by a sequence of modular multiplications. Let $R = 4^{\lceil (n+3)/2 \rceil}$ and $C = R^2\ mod N$. We use R4MG () to multiply the original input by *C* first, then all the subsequent modular multiplication can be performed without further scaling, since every intermediate output contains an extra factor of $R\ mod\ N$.

Algorithm (3): Montgomery Exponentiation algorithm [13] is shown as follows.

R4ME (M, E, N)
{
M_0 = *R4MG (M, C, N);*
$P_0 = 1$;
For $i = t - 1, t - 2 \ldots 0$; *{*
M_{i+1} = *R4MG* (M_i , M_i , *N*) ;
If ($e_i = 1$) P_{i+1} = *R4MG* (M_i , P_i , *N*)
Else $Pi_{+1} = P_i$;*}*
If ($P_n < 0$) $P_n = P_n + N$;
Return P_n ;
}

The main part of the computation is a loop in which modular squares and multiplications are performed. In the Montgomery exponentiation algorithm(R4ME), R4MG algorithm, is recalled $t + w\ (e) + 2$ times, where w(e) is number of 1's existed in binary representation of integer *E* and t stand for the length. The result of the loop is switched back from the residue domain to the normal representation of numbers. Note that, since we use algorithm (2) for R4ME (), the intermediate results of *M* and *P* are in the range [−N, N) during the entire exponentiation process, thus we only convert the final result to the range [0, N), by adding *N* to *P* if $P < 0$ ($P_n = P_n + N$). The intermediate values are listed in Table II.

TABLE II. INTERMEDIATE VARIABLE OF PROCEDURES

I	0	1	2	…	n
M_i	$M.R$	$M^2.R$	$M^4.R$	…	$M^{2^n}.R$
P_i	1	M^{e_0}	$M^{2e_1+e_0}$	…	M^E

VI. MODULAR REDUCTION

As it was discussed in the previous sections, we need to calculate reduction in such cryptosystem like RSA**.** Classical methods are so slow and need a plenty of area on FPGA. In 1998 Koc et.al proposed a very fast algorithm to calculate reduction [9] which used CSA and sign bit estimation technique. This algorithm is very suitable for hardware implementation and is faster than the others because it used *CSA* to prevent carry propagation and also a 4-bit addition is required to estimate the sign of the intermediate results. Since *CSA* is a result of adding sum and carry, and sign of a number is defined by *MSB* of it, thus in *CSA* technique, sign bit is not clear and should be calculated. In the worst case, we should add two binary numbers, carry and sum, to define sign bit. This adding should be done several times, and each times it takes a long time (*O (N) for CPA and O (logn) for CLA)*, so it is not popular to do so. For solving this problem, we should estimate sign bit instead of exactly specifying of it and at the last step we can specify sign bit exactly. In [9] it was assumed that *MSB* of *M* is 1. Hence, for arbitrary *n-bit N*, it is required to move the sign estimation position depending on the *MSB* position of *N*. This could be very inefficient in hardware implementation because the modules *N* are usually of hundreds or thousands in bits in RSA cryptosystem. Let us assume $X=\alpha N+\beta$ where $\alpha \geq 0$, $0 \leq \beta \leq N$ $X2^l \bmod N = (\alpha N+\beta)\, 2^l \bmod N = \beta 2^l$.
Now the reduction results $X \bmod N$ (=β) can be just obtained by *l*-bit right shift of $\beta 2^l$. Therefore, adding shift and count operations to the original algorithm, we simply make the algorithm to work for arbitrary *n-bit N* as in the algorithm below.

Algorithm (4)**:** The modular reduction algorithm with sign estimation [9] is shown as follows:

Inputs: $N = \sum_{i=0}^{n-1} n_i 2^i \qquad X = \sum_{i=0}^{k+n-1} x_i 2^i$

Output: $X \bmod N$

1. $t = n\text{-}2$, $S := 2^{-k-1}X$, $C = 0$, $l = 0$
2. *While* $N_{n-1} = 0$ *do begin*
   $N = 2N$, $l = l+1$
3. *End*
4. *For* $i = 1,2, \ldots, K+1$ *do begin:*
   1.4. If *ES(S, C) = (+) then*
   (C, S) = CSA $(2S + X_{k-i+1}, 2C, -N)$
   2.4. *If ES(S, C) = (–) then*
   (C, S) = CSA $(2S + X_{k-i+1}, 2C, +N)$
   3.4. If ES(S, C) = (±) then
   (C, S) = CSA $(2S + X_{k-i+1}, 2C)$
5. $R = S+C$
6. *IF* $R < 0$ *then* $R = R + N$
7. *While* $l \neq 0$
   $R = R/2$, $l = l - 1$
8. *End*
9. *Return R*

In this algorithm, *ES(S, C)* denotes Koc et al.'s main idea, the estimated sign function which uses only 4-bit addition to estimate the sign of the intermediate result. For a more detailed explanation, please refer to [9]. This algorithm is consisting of *K+1* reduction steps. The reduction steps use carry save addition. Steps 2, 3 (shift left of *N* and count up of operations) and steps 7, 8 (shift right of S and count down of operation) are additionally added to the main algorithm in [9] and also the number of iterations has been changed to *k+l+1* from *k+1*. The reduction iteration number *k+l+1* comes from the fact that the input number *X* is *(n+k)-bits* and the *MSB l*-bits of the n-bit number *N* are all '0's. As well as it was explained in [9] if the sum of $S_{n+1}S_nS_{n-1}S_{n-2} + C_{n+1}C_nC_{n-1}C_{n-2}$ regardless off carry is equal to << 1111 >>, << *positive* >> and << negative >>, then estimation function *ES(S,C)* would be respectively(±)(+)(–). Note that in all iterations of above algorithm *S+C*, *S* and *C* is in the range of $[-M, M) \subseteq (-2^n, 2^n)$.

VII. HARDWARE STRUCTURE OF MONTGOMERY'S MULTIPLICATION

A fully architecture has been designed to implement Montgomery modular multiplier using *CSA* and Booth's technique. In order to prevent delay, we use *CSA* and Booth's algorithm in radix-4. Figure2 illustrates the architecture of the radix-4 Montgomery modular multiplier based on algorithm (2). It mainly consists of an (n+2)-bit and (n+1)-bit *CSA* and also *SELTi*, *SELBi* and *SELNi* blocks. The *SELBi* block generates the values of B_i, *negb* and $sgnB_i$ ($sgnB_i$ is used for the implementation of the Roorda algorithm). *SELTi* block is used to determine (t_{i1}, t_{i0}) (step 2.3 from algorithm2). This block can be implemented by a *1FA+ 2Xors*. *SELNi* block is used for implementation of steps 2.4 and 2.5 from algorithm (2) and generation of *sgnNi* ($sgnN_i$ is used for the implementation of the Roorda algorithm). This block is implemented by several gates and *MUX*. As well as discussed in the Section V, in Booth's technique every partial product has two bits offset to left rather to former partial product. In order to do so, we used two shift registers, with variable inputs; $sgn\,Ni's$ and $sgn\,B_i's$. In step 2.8 from algorithm (2) shifting to right is used instead of dividing. Since two right bits of *(S+C)* are 0 whereas is not the reason to think two right bits of *C* and *S* are 0, thus we use *CARRY* block and C_{in} parameter. At the last of our hardware architecture, in order to transform four *n-bit* operands in less than 10ns into two n-bit operands, two *CSA* was used. To reduce power consumption, latch block is used to load the final value to the output, in the last iteration.

VIII. HARDWARE STRUCTURE OF MODULAR REDUCTION

We present hardware architecture of a high speed modular reduction based on algorithm (4). Figure3 illustrates the architecture of the modular reducer. For arbitrary *(n+k)-bit* and n-bit positive integer *X* and *N*, this hardware can calculate *X mod N*, and as depicted in figure2, it consists of two *CSA* and three signals (*N_REG*, *S_REG*, *C_REG*) and

some control logics. Once the operation starts, the *N_REG* signal which stores the inputs *N* is left circular shifted until the *MSB* of the signal is 1. (Step 2 from algorithm (4)). The two signals *S_REG* and *L_REG*, store the values $2^{k-1}x$ and 0 each at first, and the intermediate values of *CSA* addition are saved during ($k + l + 1$) iterations. One *CSA* is used to implement step3 from algorithm (4). Also, if the *MSB* of the summation result is 1, then the addition of the value in signal *N_reg* is done again. The final result is calculated and saved in *C_reg*. In addition, the 4-bit adder is used to calculate the 4-bit value *Y* which is used for sign estimation. The value *2C+1* is directly determined because the *LSB* of *2C* is always 0. According to sign estimation, one of the numbers of *N*, – *N* and 0 will be chosen which Two *MUX* do this job. *(k+1)*-Right bit of *X* would be store in $S_{0...-k-1}$, and in each iteration, *S* has one bit offset to left. Final adder was used for implementation of steps 5 and 6 from algorithm (4) which add *C* with S. If *MSB* = 1 then result should be added to *N* and at last, achieved number should be shifted *l* bits to right. Note that for time saving, 2 individual adders were used and always perform steps 5, 6 from algorithm (4). Selection one of these 2 numbers as an output is the duty of final *MUX*.

IX. CONTROL AND DATA PATH UNIT

Most hardware applications can be separated into a control path and a data path. The data path typically has some pipeline stages (register banks) and a huge block of combinational logic which shows regular structures. The control path, on the other hand, is mostly implemented through state encoding and therefore is relatively small compared to the data path. The control path and the data path are connected trough control signals and status signals, therefore the control unit also affects the delay in the data path. The control unit manages reading and writing data from the interface to the PCI bus and executes the square and multiply algorithm. The task of the "Interface" is to synchronize the different clock domains, read control commands and data from the PCI bus, and writes the result to the PCI bus. The "Control Unit" reacts on control commands and controls the whole computation. The hardware implementation of the RSA algorithm was also separated in a control path and a data path. It is shown in figure4.

X. CONCLUSION AND COMPARISION

We have presented architecture, for both modular multiplication and modular reduction. In computation the exponentiation, we use L-algorithm, which needs to about $2n$ and $\frac{3}{2}n$ modular multiplications, in the worth and average case, respectively. According to our designing, each modular multiplication require $\lceil (n+5)/2 \rceil$ clock cycles $+$ τ (τ is the time needed for n-bit addition). According to the algorithm (4) and hardware architecture that was presented, we need to *(n+10)* clock cycles + τ time to compute the reduction of $C = R^2 \ mod N$. If we assume that the frequency of the implemented hardware is $f = \frac{1}{T}$ then for exponentiation, we need to $n(n+6)T + (n+1)\tau$ and $\frac{3n}{4}(n+6)T + (\frac{3}{2}n+1)\tau \approx \frac{3}{4}n(n+11)T$ time in the worst and average case respectively. Hardware architecture and *VHDL* code for implementation of these algorithms was presented. We synthesized our design and showed that 1024-bit RSA encryption is performed in less than 15ms using a clock rate of 54.6 *MHz*. The results of the encryption for a 1024-bit block of data were presented, in terms of used CLBs (Configuration Logic Block) of FPGA and times. Table III, IV show encryption results in the worst and average case where n=513, 1025 and t=511, 1023 respectively. Table V shows the results with the public key $F4 = 2^{16} + 1 = 65537$. Also, in Table VI, we compare our results with other works. As well as it was shown we improved the speed up to three times faster than presented in [10] but it's hardware area was almost five times better than us. Although, using cellular array [3], improved the hardware area, but its speed was less than ours. Finally, we can say that a significant improvement was achieved in term of time. As it was shown in Table VI, in the terms of time-area (TA) our work is better than [3], but not than [10].

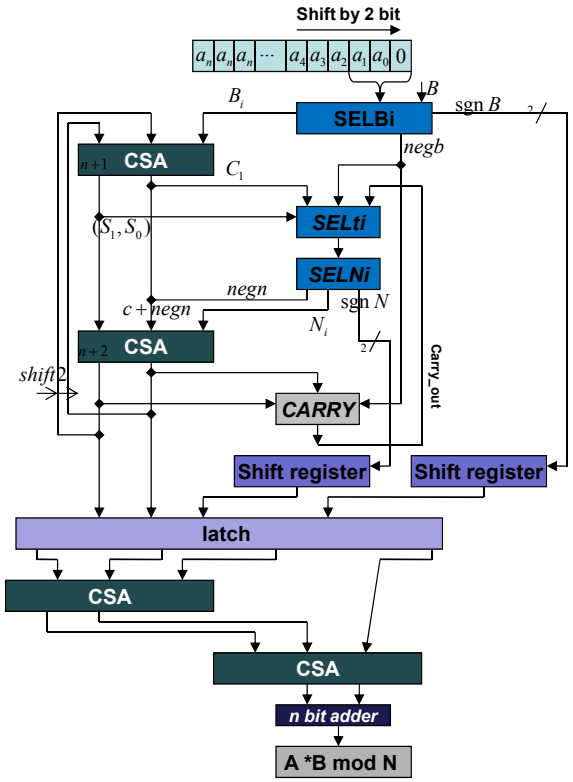

FIG.2: Hardware Architecture of Radix-4 Montgomery Multiplication Algorithm

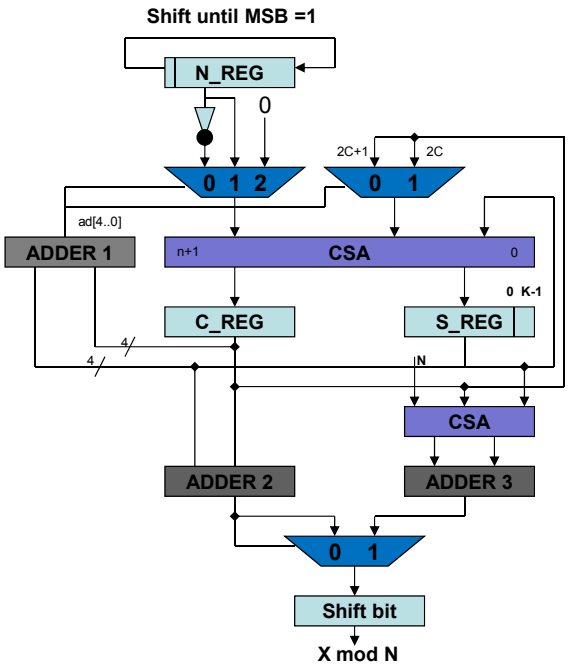

FIG.3: Hardware Architecture of Modular Reduction Algorithm

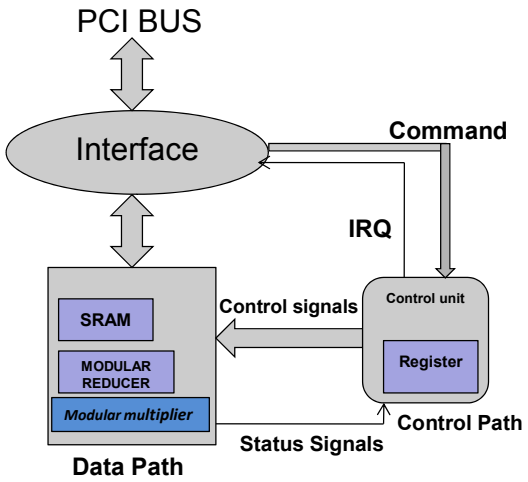

FIG.4. Separating Hardware in Control and Data Path Unit

TABLE III. APPLICATION TO RSA: ENCRYPTION FOR =512BIT

FPGA	Maximum Time for Encryption	Average Time for encryption	C(CLBs)
XC4000XL	T(ms)	T(ms)	4417 (CLBs Flip flop)
	14.982	11.326	
VirtexII	3.987	3	7251 (CLBs slices)

TABLE IV. APPLICATION TO RSA: ENCRYPTION FOR =1024 BIT

FPGA	Maximum Time for Encryption	Average Time for encryption	C(CLBs)
XC4000XL	T(ms)	T(ms)	14959 (CLBs Flip flop)
	65.89	49.467	
VirtexII	19.405	14.586	26640 (CLBs slices)

TABLE V. APPLICATION TO RSA : ENCRYPTION FOR =1024BIT WITH PUBLIC KEY =65537

FPGA	Time for encryption	C(CLBs)
XC4000XL	T(ms)	13250 (CLBs Flip flop)
	0.608	
VirtexII	0.189	24292 (CLBs slices)

TABLE VI. PERFORMANCE COMPARISON OF 1024-BIT RSA IMPLEMENTATION WITH =65537

1024 bit	*C(CLBs)*	*T(ms)*	*Time - Area*
Blum[10]	4865	0.75	3648
Ours	24292	0.189	4591
McIvor[3]	23208	0.21	4873

REFERENCES

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems " Commun. ACM, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[2] P. L. Montgomery, "Modular multiplication without trial division," Math. Comput., vol. 44, no. 7, pp. 519–521, 1985.

[3] Ciaran McIvor, Maire McLoone, John V McCanny, "Fast Montgomery Modular Multiplication and RSA Cryptographic Processor Architectures," 37th Asilomar Conference on Signals, Systems and Computers, Vol.1, No.7, pp. 379-384, Nov. 2003.

[4] Cilardo, A.Mazzeo, L. Romano, G.P. Saggese"Carry-Save Montgomery Modular Exponentiation on Reconfigurable Hardware" IEEE Proc. on DATE'04, vol.03, no. 3, pp. 206-211, 2004.

[5] T. Blum and C. Paar, "High-radix Montgomery modular exponentiation on reconfigurable hardware," IEEE Trans. Comput., vol. 50, issue 7, pp. 70-77, July 2001.

[6] Jin-Hua Hong, Cheng-Wen Wu, "Cellular-Array Modular Multiplier for Fast RSA Public-Key Cryptosystem Based on Modified Booth's Algorithm" IEEE Trans. On VLSI Systems, Vol.11, No.3, pp. 474-484, June 2003.

[7] P. Kornerup, "High-radix modular multiplication for cryptosystems," in IEEE 11th Symposium on Computer Arithmetic. 1993, pp. 277{283, IEEE Computer Society Press, Los Alamitos, CA.

[8] M. Roorda, "Method to reduce the sign bit extension in a multiplier that uses the modified booth algorithm," Electron. Lett., vol. 22, pp. 1061–1062, 1986.

[9] K.Koc, Y.Hung, "Fast algorithm for modular reduction " IEE Proc-Comput.Digit. Tech., vol. 145, no. 4, pp. 265-271, July 1998.

[10] T. Blum and C. Paar, "Montgomery modular exponentiation on reconfigurable hardware," Proc. 14th IEEE Symp. on Comp. Arith.,

[11] Walter,C.D.:'Montgomery Exponentiation Needs No Final Substraction', Electron. Lett. , 1999, 35 (21),pp.1831-1832

[12] Koc, C.K., Acar, T., Burton S. Kaliski Jr, “Analyzing and Comparing Montgomery Multiplication lgorithms,” IEEE Micro., 16(3), pp. 26-33, June 1996.

[13] Alfred J. Menezes,Paul C. van Oorschot,Scott A. Vanstone,"HANDBOOK of APPLIED CRYPTOGRAPHY," MIT Press 1996.

[14] Xilinx XC95144 CPLD family overview: http://www.xilinx.com/support/documentation/data_sheets/ds067.pdf [DS067(v5.7) may28 2009]

[15] Xilinx Virtex-II fpga family overview.
http://www.xilinx.com/support/documentation/data_sheets/ds031.pdf [DS031 (v3.5) November 5, 2007].