Computing and Informatics, Vol. 29, 2010, 849–880

# MONTGOMERY AND RNS FOR RSA HARDWARE IMPLEMENTATION

Kooroush Manochehri, Saadat Pourmozafari
Babak Sadeghian

*Department of Computer Engineering and IT*
*Amirkabir University of Technology, Tehran, Iran*
*e-mail:* {kmanochehri, saadat, basadegh}@ce.aut.ac.ir

Manuscript received 20 October 2008; revised 7 December 2009
Communicated by Liberios Vokoros

**Abstract.** There are many architectures for RSA hardware implementation which improve its performance. Two main methods for this purpose are Montgomery and RNS. These are fast methods to convert plaintext to ciphertext in RSA algorithm with hardware implementation. RNS is faster than Montgomery but it uses more area. The goal of this paper is to compare these two methods based on the speed and on the used area. For this purpose the architecture that has a better performance for each method is selected, and some modification is done to enhance their performance. This comparison can be used to select the proper method for hardware implementation in both FPGA and ASIC design.

**Keywords:** RSA, RNS, Montgomery, cryptography, CSA

## 1 INTRODUCTION

RSA [1] is the most widely used public-key cryptosystem. An RSA operation is a modular exponentiation like $c = a^e \bmod N$, which requires repeated modular multiplications. In this formula 'a' is plaintext or ciphertext and less than $N$. For security reasons RSA operand size needs to be 512-bits or more in length, hence high data throughput rates are difficult to achieve.

The Montgomery multiplication algorithm [2] is an efficient method for modular multiplication with an arbitrary modulus, particularly suitable for implementation on general-purpose computers. The method is based on an ingenious representation

of the residue class modulo N, and replaces division by $N$ operation with division by power of 2. This operation is easily accomplished on a computer or hardware design since the numbers are represented in binary form. This algorithm is the basic building block for the modular exponentiation operation, which is required in the Diffie-Hellman [3] and RSA public-key cryptosystems. Various architectures attempted to improve its performance [4, 5, 6].

Main operation of Montgomery multiplier is modular addition that consumes time for propagating carry through many bits. For this reason one can use Carry Save Adder (CSA) for avoiding long carry propagation [7].

Like Montgomery, RNS is another method for implementing RSA with parallel architecture. So the processing speed is increased, while it uses more area in hardware implementation.

Residue Number System (RNS) has long been considered an alternative to weighted (binary) representation in digital signal processing [8], public key cryptography and especially in RSA implementations [9, 10, 11]. RNS can be used to represent numbers using independent residues of manageable word length, as well as to exploit the independence of these residues in order to facilitate parallel computation of public key cryptosystems, as these typically require modular multiplication of very large integers.

In RNS, numbers are represented according to a base $\beta = (m_1, m_2, \ldots, m_k)$ that all of the moduli are prime of each others, $\gcd(m_i, m_j) = 1, i \neq j$, where $k$ is the number of elements in this base and is called the size of base. An integer 'a' is represented by the sequence $(a_1, a_2, \ldots, a_k)$ of positive integers, where $a_i = a \bmod m_i$, $i = 1, \ldots, k$. The Chinese Remainder Theorem (CRT) ensures the uniqueness of this representation within the range $0 \leq a < M$, where $M = \prod_{i=1}^{k} m_i$[12]. The proof of this theorem can be used to convert back the numbers from its residue representation:

$$a = \sum_{i=1}^{k} a_i M_i \left| M_i^{-1} \right|_{m_i} \bmod M \tag{1}$$

where $M_i = \frac{M}{m_i}$ and $\left| M_i^{-1} \right|_{m_i}$ is the inverse of $M_i$ modulo $m_i$.

The most important advantage of RNS is that addition, subtraction and multiplication are very simple and can be implemented in constant time on a parallel architecture. If $a$ and $b$ are given in their RNS form $(a_1, a_2, \ldots, a_k)$ and $(b_1, b_2, \ldots, b_k)$, the result of the mentioned operations are

$$a \pm b = \left( |a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k} \right) \tag{2}$$

$$a \times b = \left( |a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k} \right). \tag{3}$$

The disadvantages of RNS representation are twofold. First, one can not easily figure out $a(a_1, a_2, \ldots, a_k)$ is greater than $b(b_1, b_2, \ldots, b_k)$ or vise versa, and whether the overflow has occurred during the computation or not. Thus, division is difficult in this representation.

The above mentioned facts are not a drawback for cryptographic implementations including public key cryptography. In public key cryptography most of the algorithms perform the computations in a finite field or ring, which eliminates the overflow problem. Moreover, they do not require comparisons. Modular reduction (the computation of ($a$ mod $N$), multiplication ($ab$ mod $N$) and exponentiation ($a^b$ mod $N$) are the most important operations. They can be efficiently computed without division using Montgomery's algorithm.

The moduli in the form of $2^n + 1$ and $2^n - 1$ are frequently used in RNS [13] because the subtraction, addition and multiplication can be performed simply in these residues. Also calculating residue of a number in these moduli is simple, in $2^n - 1$ [14] and in $2^n + 1$ if exploiting the diminished-1 representation [15], by a simple addition the result can be obtained. Modular multiplication is the one of the operations that uses the RNS for increasing its speed.

If the numbers were in the diminished-1 representation, with Wallace tree [16] and CSA architecture [17], the modular multiplication in modulo $2^n+1$ can be done in the most efficient method [17]. Also for modular multiplication in modulo $2^n - 1$ this architecture can be used without conversion to diminished-1.

In this paper, to compare these two methods of RSA implementation the best of their implementation, according to their performance, is selected and some modification is done to enhance their performance.

This paper is organized as follows: In Section 2, the Montgomery algorithm and its radix-2 version are described. In Section 3, our modified version of Montgomery algorithm to enhance its performance is proposed. Section 4 discusses the five-to-two architecture of Montgomery multiplier that has the best-reported performance. The modified version of Montgomery multiplier that uses five-to-two architecture with its results is presented in Section 5. Proposed RSA architecture and the results are presented in Section 6. Section 7 briefly describes the Bajard RNS method for modular multiplication. RSA implementation with Bajard method is summarized in Section 8. Section 9 does some modification in Bajard algorithm to improve its performance. A main processing unit for RNS implementation is proposed in Section 10. The result of RSA implementation with RNS is shown in Section 11. Section 12 compares two methods based on the used area and the processing speed. Finally the paper is concluded in Section 13.

## 2 MONTGOMERY MODULAR MULTIPLICATION

Let the modulus $N$ be an $n$-bit integer number, i.e. $2^{n-1} \leq N < 2^n$, and let $r$ be $2^n$ [18]. The Montgomery multiplication algorithm requires that $r$ and $N$ be prime of each other, i.e., $gcd(r, N) = gcd(2^n, N) = 1$. This requirement is satisfied if $N$ is odd. In order to describe the Montgomery multiplication algorithm, first define the $N$-residue of an integer $a < N$ as $a' = ar \mod N$. Given two $N$-residues $a'$ and $b'$, the Montgomery product is defined as the $N$-residue $c' = a'b'r^{-1} \mod N$, where $r^{-1}$ is the inverse of $r$ modulo $N$, i.e., it is the number with the property $r^{-1}r = 1 \mod N$.

The resulting number $c'$ is the $N$-residue of the product $c = ab \bmod N$, since

$$c' = a'b'r^{-1} \bmod N = ar\, br\, r^{-1} \bmod N = cr \bmod N. \tag{4}$$

In order to describe the Montgomery reduction algorithm, an additional quantity, $N'$, is needed which is the integer with the property, $rr^{-1} - NN' = 1$. The integers $r^{-1}$ and $N'$ can both be computed by the extended Euclidean algorithm [19]. The computation of Monpro$(a', b')$ is achieved as follows:

**Monpro**$(\mathbf{a}', \mathbf{b}')$
$t = a'b'$;
$u = (t + (tN' \bmod r)n)/r$;
if $u \geq N$ then return $u - N$ else return $u$;

Output of this algorithm is $a'b'r^{-1} = ar\, br\, r^{-1} = abr \bmod N$. So if its arguments were $a$ and $b$, this output was $abr^{-1} \bmod N$.

The following exponentiation algorithm is one way to compute $x = a^e \bmod N$ by using Montgomery multiplication algorithm [5, 18, 20, 21]. $n$ is the number of exponent bits. Note that Monpro$(x', 1) = x'1r^{-1} = xrr^{-1} = x \bmod N$.

**ModExp**$(\mathbf{a}, \mathbf{e}, \mathbf{n})$
$a' = ar \bmod N$;
$x' = 1r \bmod N$;
for $i$ in $n - 1$ downto 0 loop
  $x' =$ Monpro$(x', x')$;
  if $e_i = 1$ then $x' =$ Monpro$(x', a')$;
end loop;
$x =$ Monpro$(x', 1)$;
return $x$;

The radix-2 version of Montgomery multiplication algorithm [22] that calculates the Montgomery product of $a$ and $b$ is summarized in the pseudo code below.

**Montgomery Multiplication**$(\mathbf{a}, \mathbf{b}, \mathbf{N})$
$S[0] = 0$;
for $i$ in 0 to $n - 1$ loop
  $q_i = (S[i]_0 + a_i b_0) \bmod 2$;
  $S[i + 1] = (S[i] + a_i b + q_i N)$ div 2;
end loop;
return $S[n]$;

$a$ and $b$ must be equal to or less than $2N$ and the output is $ab2^{-n} \bmod N$. Note that $r$ in this algorithm is equal to $2^n$.

This algorithm is more suitable for hardware implementations [23, 24, 25] because it uses modulus and division by 2 that could be implemented easier. In this algorithm arguments can be larger than $N$ and equal to or less than $2N$ [22, 23, 26] but when it is used for RSA these arguments shall never be larger than $N$ due to its property [20, 27].

The critical delay of this algorithm occurs during the calculation of the $S$ values given by the three inputs addition $S[i+1] = (S[i] + a_i b + q_i N)$.

The carry propagation resulting from the very large operand additions is the main contributing factor for the delay.

## 3 SIMPLE RADIX-2 MONTGOMERY MULTIPLIER

As mentioned, calculation of summation is the critical delay of radix-2 Montgomery multiplier. Using the CSA architecture can eliminate this delay but one parameter of this delay is to calculate $q_i$ and multiply with $N$ and then to add the result to summation of the two other parameters.

In this algorithm in order to calculate $q_i$, first bit of previous result is added by $a_i b_0$. So if one can make $b_0$ equal to zero this step can be removed. Assuming $b$ as $2b$ can do this, because the first bit of $2b$ is zero and $a_i b_0$ is equal to zero. This assumption for some systems such as RSA can be used because inputs of Montgomery algorithm can be larger than $N$ and if $b$ is less than $N$ (for some systems) then $2b$ is less than $2N$ and satisfies the Montgomery multiplier requirement.

The algorithm required one extra clock cycle by adding one bit to $b$. So new algorithm can be rewritten as follows [28]:

**New Montgomery Multiplication**$(\mathbf{a}, \mathbf{b}, \mathbf{N})$
$S[0] = 0$;
$B\_\text{new} = 2b$;
for $i$ in 0 to $n$ loop
  $S[i+1] = (S[i] + a_i B\_\text{new} + S[i]_0 N)$ div 2;
end loop;
return $S[n+1]$;

New multiplier assumes that the last bit of $a$ and $N$ is zero so $a_n$ in this algorithm is zero. For the result only first $n$ bits of $S[n+1]$ should be returned.

The modified algorithm uses $2b$ instead of $b$ as input and with one extra division by two, the result is $a\,2b\,2^{-(n+1)} = a\,2b\,2^{-1}2^{-n} = ab2^{-n} \bmod N$.

This is the required result. So this algorithm can be used instead of Radix-2 Montgomery multiplier in exponentiation algorithm without any change in that algorithm.

For instance the new algorithm computes Montgomery multiplication $(a)1101 \times (b)1001 \bmod 1111 = 1100$ as follows:

| | S[0]=0<br>B_new=10010 |
|---|---|
| $i = 0$ | $S[1] = 00000 + 1 \times 10010 + 0 \times 1111$ div $2 = 01001$ |
| $i = 1$ | $S[2] = 01001 + 0 \times 10010 + 1 \times 1111$ div $2 = 01100$ |
| $i = 2$ | $S[3] = 01100 + 1 \times 10010 + 0 \times 1111$ div $2 = 01111$ |
| $i = 3$ | $S[4] = 01111 + 1 \times 10010 + 1 \times 1111$ div $2 = 11000$ |
| $i = 4$ | $S[5] = 11000 + 0 \times 10010 + 0 \times 1111$ div $2 = 01100$ |

## 4 FIVE-TO-TWO CSA ARCHITECTURE

The main calculations of CSA are

$$\text{sum} = x1 \text{ XOR } x2 \text{ XOR } x3 \tag{5}$$

$$\text{carry} = (x1 \text{ AND } x2) \text{ OR } (x1 \text{ AND } x3) \text{ OR } (x2 \text{ AND } x3). \tag{6}$$

The sum of bit vectors sum and carry is equal to the sum of the three input bit vectors $x1$, $x2$ and $x3$. The benefit of this architecture is that no carry propagation is needed and therefore fast summation is possible.

An outline diagram of five-to-two CSA operation [7, 20] is shown in Figure 1. Montgomery algorithm can be changed for this architecture as follow:

**Five-to-two CSA Montgomery Multiplication**$(\mathbf{a1}, \mathbf{a2}, \mathbf{b1}, \mathbf{b2}, \mathbf{N})$
$S1[0] = 0;$
$S2[0] = 0;$
for $i$ in 0 to $n-1$ loop
  $q_i = (S1[i]_0 + S2[i]_0 + (a_i(b1_0 + b2_0))) \bmod 2;$
  $S1[i+1], s2[i+1] = CSR(S1[i] + S2[i] + a_i(b1 + b2) + q_i N) \text{ div } 2;$
end loop;
return $S1[n], s2[n];$

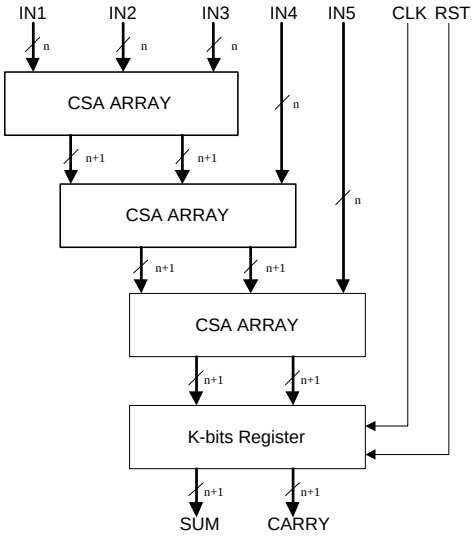



Fig. 1. Block diagram of five-to-two CSA

Note that the input operands $a$ and $b$ and the output product $S$ are represented in carry save format as $a1$ and $a2$, $b1$ and $b2$, and $S1$ and $S2$, respectively. CSR stands for carry save representation.

Barrel Register Full Adder (BRFA) can be used to determine $a_i$ as shown in Figure 2. So there is no need to compute full addition of the input operands $a1$ and $a2$ [29].

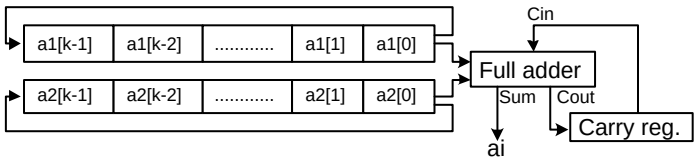



Fig. 2. Barrel register full adder diagram

For this architecture at the beginning of each multiplication one extra clock cycle is required to reset the signal $S1[0]$ and $S2[0]$ to zero. Thus this algorithm can be executed in only $n + 1$ clock cycles.

## 5 IMPLEMENTATION OF MODIFIED MONTGOMERY MULTIPLICATION ALGORITHM

To implement the new algorithm, CSA architecture is used. The delay of calculating $q_i$ and then multiplying with $N$ due to modified algorithm is removed. The five to two CSA algorithm can be rewritten as follows [28]:

**Five-to-two CSA New Montgomery**
Multiplication($\mathbf{a1}, \mathbf{a2}, \mathbf{b1}, \mathbf{b2}, \mathbf{N}$)
$S1[0] = 0$;
$S2[0] = 0$;
for $i$ in 0 to $n$ loop
  $Si_0 = (S1[i]_0 + S2[i]_0) \bmod 2$;
  $S1[i+1], s2[i+1] = CSR(S1[i] + S2[i] + a_i(b1 + b2) + Si_0 N)$ div 2;
end loop;
return $S1[n+1], s2[n+1]$;

In this implementation calculation of $a_i(b1_0 + b2_0)$ is removed but some extra memory must be used to store one added bit; but, for $a1$ and $a2$ there is no need to this extra memory because one can change BRFA to calculate $a_n$ as zero for output.

The block diagram for the new algorithm implementation is shown in Figure 3. In this figure $S1_0$ and $S2_0$ stand for $S1[i]_0$ and $S2[i]_0$ respectively in the algorithm.

In this architecture, when calculating $S1 + S2 + a_i(b1 + b2)$, the result of $N$ AND ($S1_0$ XOR $S2_0$) is prepared and there is no delay to wait for its results. In previous implementation $S1[i]_0$ XOR $S2[i]_0$ XOR ($a_i$ AND ($b1_0$ XOR $b2_0$)) must be calculated and then AND with $N$, that has more delay.

This architecture needs one clock for resetting $S1[0]$ and 4, also one extra clock cycle for internal loop. Thus this algorithm can be executed in $n + 2$ clock cycles.

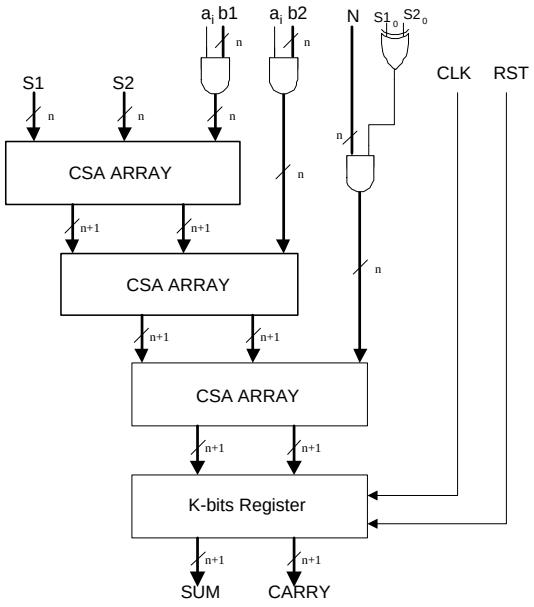



Fig. 3. Block diagram of five-to-two CSA for new algorithm

For I/O interface of both implementations, the block diagram shown in Figure 4 is used. So all reported results include these I/O registers and additional circuits. In this architecture the inputs are registered into the microchip at a rate of 32 bits per clock cycle. Likewise, the outputs are clocked out of the chip 32 bits per clock cycle. This is due to the limited number of I/O pins available.

For comparing the two algorithms, they are implemented by VHDL language and synthesized by the Leonardo Spectrum 2002 tool for both ASIC and FPGA technologies. For ASIC synthesis, CMOS 0.6 library and for FPGA synthesis, Xilinx Virtex2 series are used.

Tables 1 and 2 provide results of these implementations for CMOS 0.6 library. Tables 3 and 4 show these results for FPGAs.

| Bit Length ($n$) | Clock Speed (MHz) | Area (Gates) | Throughput Rate (Mb/s) |
|---|---|---|---|
| 512 | 101.5 | 14 312 | 101.30 |
| 1 024 | 122.1 | 28 613 | 121.98 |

Table 1. Results of ASIC synthesis of Montgomery multiplier (CMOS 0.6 library)

It should be noted that for calculating throughput rate, for standard Montgomery multiplier $n + 1$ is used and for new Montgomery multiplier $n + 2$ clock cycles are used.

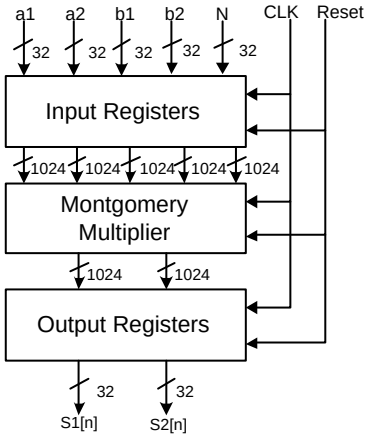



Fig. 4. I/O interface for implementations of Montgomery multiplication algorithm

| Bit Length ($n$) | Clock Speed (MHz) | Area (Gates) | Throughput Rate (Mb/s) |
|---|---|---|---|
| 512 | 146.7 | 1 4318 | 146.13 |
| 1 024 | 139.2 | 28 618 | 138.93 |

Table 2. Results of ASIC synthesis of new-Montgomery multiplier (CMOS 0.6 library)

The tables show that the new algorithm has better performance than the standard algorithm. For instance for ASIC design of 512 bits length, there are only 6 extra gates for area but throughput rate is increased to 44.25 %. For FPGA design the results and performance improved, because the area is decreased too and this is due to its structure. For instance, for 512 bits the results area decreased (2 slices) and also throughput rate is increased (45.98 %). So it is obvious that this new algorithm has greater performance in FPGA designs.

The advantage of the new algorithm can be categorized as follows:

1. Higher performance: For FPGA design it used less area and has higher frequency. For ASIC design its area increased few gates only but its throughput increased substantially, so it has better performance.
2. There is no need to change the algorithms or systems that use Montgomery multiplier because their inputs and outputs remain unchanged.

| Xilinx Device | Bit Length ($n$) | Clock Speed (MHz) | Area (Slices) | Throughput Rate (Mb/s) |
|---|---|---|---|---|
| XC2V1500 | 512 | 49.3 | 3 127 | 49.20 |
| XC2V3000 | 1 024 | 56.1 | 6 249 | 56.05 |

Table 3. Results of FPGA synthesis of Montgomery multiplier (Xilinx virtex2 library)

| Xilinx Device | Bit Length ($n$) | Clock Speed (MHz) | Area (Slices) | Throughput Rate (Mb/s) |
|---|---|---|---|---|
| XC2V1500 | 512 | 72.1 | 3 125 | 71.82 |
| XC2V3000 | 1 024 | 79.2 | 6 243 | 79.05 |

Table 4. Results of FPGA synthesis of new-Montgomery multiplier (Xilinx virtex2 library)

3. This algorithm can also be used for sequential algorithms or software implementations, because one of its steps has been removed and thus it can be run faster.

## 6 RSA HARDWARE IMPLEMENTATION WITH MONTGOMERY MULTIPLIER

An RSA operation is a modular exponentiation with operands satisfying the conditions stated previously. A well known and widely used modular exponentiation algorithm is the square and multiply algorithm given below [22, 23, 24, 25], which computes $M = c^d \bmod N$. $d_k$ is the bit length of the exponent $d$ and montmult refers to radix-2 Montgomery multiplication algorithm.

**Modular Exponentiation**$(\mathbf{c}, \mathbf{d}, \mathbf{N})$
$K = 2^{2n} \bmod N$;
$P[0] = \text{montmult}(K, c, N)$;
$R[0] = \text{montmult}(K, 1, N)$;
for $i$ in 0 to $d_k$ loop
  $P[i+1] = \text{montmult}(P[i], P[i], N)$;
  if $d[i] = 1$ then $R[i+1] = \text{montmult}(R[i], P[i], N)$;
end loop;
$M = \text{montmult}(1, R[d_k], N)$;
*return* $M$;

By using the new five-to-two CSA multiplier architecture to calculate the montmult stages of this algorithm, this algorithm can be rewritten as follows:

**New Modular Exponentiation**$(\mathbf{c}, \mathbf{d}, \mathbf{N})$
$K = 2^{2n} \bmod N$;
$P1[0], P2[0] = 5to2\_\text{new\_montmult}(K, 0, c, 0, N)$;
$R1[0], R2[0] = 5to2\_\text{new\_montmult}(K, 0, 1, 0, N)$;
for $i$ in 0 to $d_k$ loop
  $P1[i+1], P2[i+1] = 5to2\_\text{new\_montmult}(P1[i], P2[i], P1[i], P2[i], N)$;
  if $d[i] = 1$ then
    $R1[i+1], R2[i+1] = 5to2\_\text{new\_montmult}(R1[i], R2[i], P1[i], P2[i], N)$;
end loop;
$M1, M2 = 5to2\_\text{new\_montmult}(1, 0, R1[d_k], R2[d_k], N)$;
$M = M1 + M2$;
return $M$;

An analysis of the total number of clock cycles required for a full RSA operation is given in Table 5.

| Type of Operation | Corresponding Algorithm Calculation | Number of Clock Cycles |
|---|---|---|
| pre-processing | $P1[0], P2[0]$<br>$R1[0], R2[0]$ | $n+2$ |
| For Loop | $P1[i+1], P2[i+1]$<br>$R1[i+1], R2[i+1]$ | $d_k(n+2)$ |
| post-processing | $M1, M2$ | $n+2$ |
| Final Addition | $M$ | $n+2$ |
| Total Number of Clock Cycles | | $(n+2)(d_k+3)$ |

Table 5. Clock cycles to compute RSA

Pre and post processing are required to convert the operands in the algorithm to and from Montgomery format. The final addition operation makes use of the BRFA component and uses $n+2$ cycles for calculation; but if montmult is used for this purpose [30] this number is reduced to $n/3+2$. This proposed architecture is used to implement RSA using new Montgomery multiplier and the result was as shown in Tables 6 and 7.

| Bit Length ($n$) | Clock Speed (MHz) | Area (Gates) | Throughput Rate (Mb/s) |
|---|---|---|---|
| 512 | 74.1 | 38 283 | 0.29 |
| 1 024 | 70.0 | 77 711 | 0.14 |

Table 6. Results of ASIC synthesis of proposed architecture for RSA (CMOS 0.6 library)

| Xilinx Device | Bit Length ($n$) | Clock Speed (MHz) | Area (Slices) | Throughput Rate (Mb/s) |
|---|---|---|---|---|
| XC2V3000 | 512 | 72.0 | 9 047 | 0.28 |
| XC2V6000 | 1 024 | 79.5 | 18 008 | 0.15 |

Table 7. Results of FPGA synthesis of proposed architecture for RSA (Xilinx Virtex2 library)

In this implementation the length of exponent is half of the operand size. For example for operand size of 1024 bits, the exponent gets 512 bits length.

## 7 BAJARD RNS MODULAR MULTIPLIER

The Bajard method for RNS implementation is the fastest method so far [9]. This is a general method and is independent of selecting a modulus, but most of the

RNS implementations and particularly the RSA implementation by RNS used the modulus in the form of $2^n + 1$ and $2^n - 1$. So it can be modified to achieve more performance by using these moduli.

In the Bajard method, Montgomery modular multiplication algorithm is used for modular multiplication (or modular exponentiation) and implemented by residue number system. This method is introduced in [9] and summarized in this section.

This method uses two RNS bases of size $k$, $\beta = (m_1, \ldots, m_k)$ and $\beta' = (m_{k+1}, \ldots, m_{2k})$. All of the $m_i$ are prime of each other and the relationship $M' \geq M$ exists with the assumption that $M = \prod_{i=1}^{k} m_i$ and $M' = \prod_{i=k+1}^{2k} m_i$ should take place. Since this is based on Montgomery algorithm, the outputs are in the form of $abM^{-1} \bmod N$ and like Montgomery method, multiplication by $M$ gives favorite result, which is $ab \bmod N$. The Bajard RNS algorithm is as follows:

The RNS Montgomery modular multiplication algorithm
Inputs: Two RNS bases $\beta = (m_1, \ldots, m_k)$ and $\beta' = (m_{k+1}, \ldots, m_{2k})$
such that $M = \prod_{i=1}^{k} m_i < M' = \prod_{i=1}^{k} m_{k+i}$ and $\gcd(M, M') = 1$;
a redundant moduli $m_r, \gcd(m_r, m_i) = 1, i = 1 \ldots 2k$;
a positive integer $N$ represented in both RNS bases such that,
$0 < (k+2)^2 N < M < M'$ and $\gcd(N, M) = 1,\ \gcd(N, M') = 1$;
two positive integers $a, b$ represented in both RNS bases with $ab < MN$.
Outputs: A positive integer $\hat{r}$ represented in both RNS bases,
such that $\hat{r} = abm^{-1} \bmod N$ and $\hat{r} < (k+2)N$.

$MM(a, b, N)$ :
$1 - q \leftarrow (a \times b) \times (-N^{-1}) \, in \, \beta$
$2 - [q \, in \, \beta] \rightarrow [\hat{q} \, in \, \beta']$(First base extension)
$3 - \hat{r} \leftarrow (a \times b + \hat{q} \times N).M^{-1} \, in \, \beta' \, and \, m_r$
$4 - [\hat{r} \, in \, \beta] \leftarrow [\hat{r} \, in \, \beta']$(Second base extension)

Instructions in steps 1 and 3 can be performed in parallel by full RNS operations. As a consequence the complexity of the algorithm clearly relies on the two base extensions on steps 2 and 4.

If the number of multiplications was the criterion for speed, two modular multiplications in line 1 and three multiplications in line 3 are needed. Also if the number of memory for saving constant values was a criterion for area, one memory in line 1 for saving the values of $-N^{-1}$ in each modulus and two memories in line 3 for saving the values of $N$ and $M^{-1}$ in each modulus are needed.

RSA cryptosystem requires modular exponentiation, so one can use this algorithm and reuse its output each time as its input until the desired exponent is achieved.

The instruction in line 2 (first base extension) consists of converting $q$ obtained in its RNS form $(q_1, \ldots, q_k)$ in the base $\beta$ to its RNS representation in base $\beta'$. For this purpose the following relations are used. In these relationships $q_i$ represents $q \bmod m_i$.

$$\sigma_i \quad = \quad q_i \times \left| M_i^{-1} \right|_{m_i} \bmod m_i, i = 1, \ldots, k \tag{7}$$

$$\hat{q}_j = \left| \sum_{i=1}^{k} |M_i|_{m_j} \times \sigma_i \right|_{m_j}, j = k+1, \ldots, 2k \tag{8}$$

So $\sigma_i$ should be first computed and then used in next relation. In this step a modular multiplication is needed for calculating $\sigma_i$ and $k$ modular multiplications for $\hat{q}_j$. Furthermore it requires one memory for saving the value of $M_i^{-1}$ in each modulo $m_i$ and one for saving $M_i$ in each modulo $m_j$.

In line 4 (second base extension) first the value of $\xi_j$ must be computed from the following relation:

$$\xi_j = \hat{r}_j \times \left| {M'}_j^{-1} \right|_{m_j} \bmod m_j, j = k+1, \ldots, 2k \tag{9}$$

and then the desired resultant value can be computed from:

$$\hat{r}_{m_i} = \left| \sum_{j=1}^{k} \left| M'_j \right|_{m_i} \times \xi_j - |\alpha \times M'|_{m_i} \right|_{m_i} \quad i = 1, \ldots, k \tag{10}$$

where $\alpha$ is computed from:

$$\alpha = \left| \left| M'^{-1} \right|_{m_r} \times \left( \sum_{j=1}^{k} \left| M'_j \right|_{m_r} \times \xi_j - |\hat{r}|_{m_r} \right) \right|_{m_r} \tag{11}$$

For simplicity and ease of the calculation in the last equation, $m_r$ is usually assumed as power of 2 such that $m_r \geq k$, $k$ representing the number of moduli in each base set.

## 8 BAJARD RSA IMPLEMENTATION USING RNS

Implementation of RSA with Montgomery modular multiplication algorithm is as follows. The final result of this algorithm is $c = a^e \bmod N$ [5, 20].

**RSA**$(\mathbf{a}, \mathbf{e}, \mathbf{N})$
$\bar{a} = ar \bmod N$;
$\bar{c} = 1r \bmod N$;
for $i$ in $n-1$ downto 0 loop
  $\bar{c} = \text{Monpro}(\bar{c}, \bar{c})$;
  if $e_i = 1$ then $\bar{c} = \text{Monpro}(\bar{a}, \bar{c})$;
end loop;
$c = \text{Monpro}(\bar{c}, 1)$;
return $c$;

Monpro represents the Montgomery modular multiplication algorithm and $r$ represents the auxiliary moduli. The number of bits in $N$ (or $e$) is shown in this algorithm by $n$.

The rule for this algorithm is that the input is converted to Montgomery representation and then according to the number of bits of exponent the internal loop is repeated in order to calculate the desired modular exponentiation and finally the result of this loop is converted back to binary representation by the last Montgomery modular multiplication.

Note that in this algorithm 'a' represents the plaintext that must be encrypted. As mentioned before, RNS final result is a single modular multiplication where in RSA a modular exponentiation is needed.

Since the Bajard RNS method is based on Montgomery algorithm, the previous RSA algorithm can be used except for Monpro, the RNS Montgomery modular multiplication algorithm ($MM(a, b, N)$) must be substituted and redundant residue ($r$) should be substituted by $M$ (multiplications of modulus).

By this substitution the final result is less than $(k + 2)N$, since it must be less than $N$ so this difference must be corrected. The solution that is proposed by Bajard [9] is that since this error is obtained by the calculation of the first base extension approximately, for the last call of multiplication algorithm (that is used for convert back to binary representation) this base extension should be calculated exactly. The Mixed Radix System (MRS) [31] can be used for this exact base extension. The relations of MRS are as below:

$$|q|_{m_j} = |t_1 + t_2 m_1 + t_3 m_1 m_2 + \ldots + t_k m_1 \ldots m_{k-1}|_{m_j} . \tag{12}$$

In other words in this system the weight of each number $t_i$ is 1, $m_1, m_1 m_2, \ldots,$ $m_1 m_2 m_3 \ldots m_k$. The value of $t_i$ can be computed as follows:

$$\begin{aligned}
&t_1 = q \bmod m_1 = q_1\\
&t_2 = (q_2 - t_1)c_{12} \bmod m_2;\\
&\vdots\\
&t_k = (\ldots((q_k - t_1)c_k - t_2)c_{2k} - \ldots - t_{k-1})c_{(k-1)k} \bmod m_k \bmod m_2;
\end{aligned}$$

$c_{ij}$ represents $m_i^{-1} \bmod m_j$ and $q_i$ is the value from step 1 of the RNS algorithm which is equal to $q_i = q \bmod m_i$.

## 9 MODIFIED BAJARD ALGORITHM WITH IMPROVED PERFORMANCE

The Bajard algorithm uses less clock cycles and has more speed for calculation for RNS implementation than any other methods. In this section Bajard algorithm is used and optimized for RSA hardware implementation to reduce the number of clock cycles and the area.

The values calculated during internal calculations that are not necessary for RSA final results are combined with other calculations to reduce the calculation time, as well as the area. For further clarity the optimizations for the main values are presented as follows.

### 9.1 Calculation of $\sigma_i$ in the First Base Extension

As shown in the first base extension of the algorithm, it receives the value of $q = (a \times b) \times (-N^{-1})$ in the RNS representation in base $\beta$ and calculates the value of $\hat{q}$. So in RNS algorithm the value of $q$ is not required and can be combined with the first base extension to reduce calculation cycles. The new relationship for $\sigma_i$ is as follows:

$$\begin{aligned} \sigma_i &= q_i \times \left|M_i^{-1}\right| \bmod m_i = \left(a \times b \times \left|-N^{-1}\right|_{m_i}\right) \times \left|M_i^{-1}\right|_{m_i} \bmod m_i \\ &= (a \times b) \times \left|-N^{-1} \times M_i^{-1}\right|_{m_i} \end{aligned} \tag{13}$$

In order to calculate $a \times b$, then multiply by $M_i^{-1}$, then multiply the final result by $-N^{-1}$ and assuming each multiplication requires one cycle, at least 3 cycles are required. By combined relation, these cycles reduce to 2 cycles. As a result the speed is increased and area decreased, because in the new relation the value of $\left|-N^{-1} \times M_i^{-1}\right|_{m_i}$ should be saved instead of saving two values $\left|M_i^{-1}\right|_{m_i}$ and $|-N^{-1}|_{m_i}$.

### 9.2 Calculation of $\xi_i$

The values of $\xi_j$ play an important role in the calculation of the final results. The equations that are required to calculate $\xi_j$ are listed below.

$$\xi_j = \hat{r}_j \times \left|M'^{-1}_j\right|_{m_j} \bmod m_j \tag{14}$$

$$\hat{r}_j = (a \times b + \hat{q}_j \times N) \times M^{-1} \bmod m_j \tag{15}$$

$$\hat{q}_j = \left(\sum_{i=1}^{k} |M_i|_{m_j} \times \sigma_i\right) \bmod m_j \tag{16}$$

If one starts with the first equation then the values are substituted and it implies that:

$$\begin{aligned} \xi_j &= \left((a \times b + \hat{q}_j \times N) \times M^{-1}\right) \times \left|M'^{-1}_j\right|_{m_j} \bmod m_j \\ &= \left(a \times b \times \left|M^{-1} \times M'^{-1}_j\right|_{m_j}\right) + \left(\hat{q}_j \times \left|N \times M^{-1} \times M'^{-1}_j\right|_{m_j}\right) \bmod m_j \\ &= \left(a \times b \times \left|M^{-1} \times M'^{-1}_j\right|_{m_j}\right) \\ &\quad + \left(\left|\sum_{i=1}^{k} |M_i|_{m_j} \times \sigma_i\right|_{m_j} \times \left|N \times M^{-1} \times M'^{-1}_j\right|_{m_j}\right) \bmod m_j \end{aligned}$$

$$= \underbrace{\left(a \times b \times \left|M^{-1} \times M'^{-1}_{j}\right|_{m_j}\right)}_{\sigma_j} + \underbrace{\sum_{i=1}^{k}\left(\sigma_i \times \left|M_i \times N \times M^{-1} \times M'^{-1}_{j}\right|_{m_j}\right)}_{\delta_j} \bmod m_j$$

In this relationship the first multiplication is named $\sigma_j$ and the second one is named $\delta_j$. The calculation of $\sigma_j$ and the calculation of $\sigma_i$ can be computed in parallel; as a result the computation time is reduced. Also the area is reduced due to combining the fixed values, as a result instead of saving four values $|M_i|_{m_j}$,$|N|_{m_j}$,$|M^{-1}|_{m_j}$ and $\left|M'^{-1}_{j}\right|_{m_j}$just two values $\left|M^{-1} \times M'^{-1}_{j}\right|_{m_j}$ and $\left|M_i \times N \times M^{-1} \times M'^{-1}_{j}\right|_{m_j}$ should be saved. Furthermore, instead of $k+4$ multiplication cycles ($k$ multiplication cycles for calculation of sigma for $\hat{q}_j$ and 4 multiplication cycles for other calculations) only $k+2$ cycles ($k$ multiplication cycles for $\delta_j$ and two cycles for $\sigma_j$) are required.

### 9.3 Calculation of $|\hat{\mathbf{r}}|_{\mathbf{m_r}}$

Since the value of $|\hat{r}|_{m_r}$ is required for calculation of $\alpha$, its value should be computed during calculation of the other values. The formula for this computation is as follows:

$$|\hat{r}|_{m_r} = (a \times b + \hat{q} \times N) \times M^{-1} \bmod m_r \tag{17}$$

$\hat{q}$ can be computed by the relation $\hat{q}_r = \sum_{i=1}^{k} |M_i|_{m_r} \times \sigma_i \bmod m_r$. So with substitution in $|\hat{r}|_{m_r}$, the new formula is as follows:

$$|\hat{r}|_{m_r} = \underbrace{\left(a \times b \times M^{-1}\right)}_{\sigma_r} + \underbrace{\sum_{i=1}^{k} \sigma_i \times \left|M_i \times N \times M^{-1}\right|_{m_r}}_{\delta_r} \bmod m_r.$$

In this equation the first term is named $\sigma_r$ and the second one is named $\delta_r$. The calculation of $\sigma_r$ can be done while calculating $\sigma_i$. Due to this combining, the number of multiplications is reduced from k+3 (k multiplications for calculation of $\hat{q}_r$ and 3 for $|\hat{r}|_{m_r}$ calculation) to $k+1$ ($k$ multiplication cycles for $\delta_r$ and one for $\sigma_r$ calculation); furthermore the number of memory for saving the constant value is reduced from 3 to 2.

After computing $|\hat{r}|_{m_r}$ the value of $\alpha$ can be computed as follows:

$$\begin{aligned} \alpha &= \left|M'^{-1}\right|_{m_r} \times \left(\sum_{j=1}^{k} |M'_j|_{m_r} \times \xi_j - |\hat{r}|_{m_r}\right) \bmod m_r \\ &= \underbrace{\sum_{j=1}^{k} \xi_j \times \left|M'^{-1} \times M'_j\right|_{m_r}}_{\alpha_1} - |\hat{r}|_{m_r} \times \left|M'^{-1}\right|_{m_r} \bmod m_r \end{aligned}$$

In this equation the first term is named $\alpha_1$ for individual calculations.

### 9.4 Calculation of $|\hat{r}|$ in the Final Result

After the calculations of previous values now the final result or the output of algorithm ($\hat{r}$) is obtained. Due to removing some intermediate values for optimization, the values of $|\hat{r}|_{m_j}$ should be computed, and $|\hat{r}|_{m_i}$ should be computed. As a result the following relations can be used:

$$\begin{aligned} |\hat{r}|_{m_i} &= \underbrace{\sum_{j=1}^{k} |M'_j|_{m_i} \times \xi_j}_{\rho} - |\alpha \times M'|_{m_i} \mod m_i \\ |\hat{r}|_{m_j} &= \xi_j \times |M'_j| \mod m_j. \end{aligned}$$

The first term in $|\hat{r}|_{m_i}$ is named $\rho$. The relation of $|\hat{r}|_{m_j}$ is obtained from the fact that the value of $\xi_j$ is equal to $\hat{r}_j \times \left|M'^{-1}_j\right|_{m_j} \mod m_j$ and as a result $|\hat{r}|_{m_j}$ is produced by multiplying it with the value of $|M'_j|_{m_j}$. With this method one redundant multiplication is added but due to reducing the multiplication cycles obtained by other optimizations, this overhead can be omitted and overall reduction in multiplication cycles is achieved.

If the numbers of clock cycles are mentioned, all calculations required for RNS method can be categorized as follows:

1.

$$\begin{aligned} \sigma_i &= (a \times b) \times \left|-N^{-1} \times M_i^{-1}\right|_{m_i} \mod m_i\,,\ i = 1, \ldots, k \\ \sigma_j &= (a \times b) \times \left|M^{-1} \times M'^{-1}_j\right|_{m_j} \mod m_j\,,\ j = k+1, \ldots, 2k \\ \sigma_r &= (a \times b) \times M^{-1} \mod m_r \end{aligned}$$

2.

$$\begin{aligned} \xi_j &= \sigma_j + \sum_{i=1}^{k} \left(\sigma_i \times \left|M_i \times N \times M^{-1} \times M'^{-1}_j\right|_{m_j}\right) \mod m_j \\ \hat{r}_{m_r} &= \sigma_r + \sum_{i=1}^{k} \sigma_i \times \left|M_i \times N \times M^{-1}\right|_{m_r} \mod m_r \end{aligned}$$

3.

$$\begin{aligned} \rho &= \sum_{j=1}^{k} |M'_j|_{m_i} \times xi_j \mod m_i \\ \alpha_1 &= \sum_{j=1}^{k} \xi_j \times \left|M'^{-1} \times M'_j\right|_{m_r} \mod m_r \end{aligned}$$

4.

$$\alpha = \alpha_1 - |\hat{r}|_{m_r} \times \left|M'^{-1}\right|_{m_r} \bmod m_r$$

5.

$$\begin{aligned} |\hat{r}|_{m_i} &= \rho - |\alpha \times M'|_{m_i} \bmod m_i \\ |\hat{r}|_{m_j} &= \xi_j \times |M'_j|_{m_j} \, mod \, m_j \end{aligned}$$

In this categorization the relations in each category can be computed in parallel because they are independent of each other. Due to equal number of multiplication and summation in each category, the relations in one category are finished in the same cycles and their results can be used in the next category.

It is obvious that in the first category 3 multiplications, in the second category $k$ multiplications and one addition, in the third category $k$ multiplications, in the fourth category one multiplication and one subtraction, in the fifth category $k$ multiplications and for $|\hat{r}|_{m_i}$ a redundant subtraction are needed. Note that $k$ is the number of moduli in base $\beta$ or $\beta'$.

## 10 DESIGN OF THE MAIN RNS PROCESSING UNITS WITH THE BEST PERFORMANCE

As mentioned before, the modulus in the form of $2^n - 1$ and $2^n + 1$ is more useful and cost effective for RNS implementations. For selecting these moduli from the set $2^{n_1} - 1, 2^{n_1} + 1, 2^{n_2} - 1, 2^{n_2} + 1, \ldots, 2^{n_L} - 1, 2^{n_L} + 1$ one should notice that all of these moduli are prime of each other. For this purpose due to the proof that is presented in [13] all the $n_i$, $i = 1, , L$ must be prime of each other; and for having a similar modulus length, these parameters should be as close as possible.

The goal of this section is to improve the Bajard method for RSA hardware implementation by using the above-mentioned moduli and implementing this improved method by proper architectures for each main processing unit. In designing each unit the performance is mentioned.

For ease of implementation and for speed improvement, modulus in the form of $2^n - 1$ is selected as first base set ($M$) and modulus in the form of $2^n + 1$ is selected as second base set ($M'$). This assumption satisfies the relation $M < M'$. Moreover, by this selection the decision-making for hardware implementation of each step in the RNS algorithm can be done easily.

### 10.1 $2^n + 1$ Modular Multiplier

$2^n + 1$ modular multiplication has the main role in RNS implementation but operations in these moduli are not as easy as those in $2^n - 1$ [14].

The diminished-1 representation of numbers was proposed by Leibowitz [15], as a convenient and efficient form for modulo $2^n + 1$ operations on binary numbers. If $d(a)$ is the diminished-1 representation of 'a' then:

$$d(a) = a - 1 \bmod (2^n + 1). \tag{18}$$

The advantage of this representation is that zero is uniquely identified by the most significant bit (MSB = 1), for which all arithmetic operations are inhibited.

There are the following relationships for arithmetic operations within the representation:

$$\begin{aligned} d(a+b) &= d(a) \oplus d(b) = d(a) + d(b) + 1 \bmod (2^n + 1) && (19)\\ d(a-b) &= d(a) \oplus [-d(b)] = d(a) + \overline{d(b)} + 1 \bmod (2^n + 1) && (20)\\ d\left(\sum_{i=1}^{k} a_i\right) &= \sum_{i=1}^{k} \oplus d(a_i) = d(a_1) \oplus \ldots \oplus d(a_n) \\ &= d(a_1) + \ldots + d(a_n) + n - 1 \bmod (2^n + 1). && (21) \end{aligned}$$

In these relationships, $\oplus$ represents addition and $[-x]$ represents negative of number $x$ in diminished-1 format. $\overline{d(b)}$ represents the one's complement of $d(b)$ and $\sum \oplus d(a_k)$ used for summation of the numbers $d(a_k)$ in modulo $2^n+1$ in diminished-1 form.

In [17], by using the Wallace tree and diminished-1 representation, an efficient method for modular multiplication modulo $2^n+1$ is presented. By using this method there is no need to expand the adders to $2n$ bits for $n$ bit numbers. The multiplication is done by a series of addition that in each addition step, carry should be complemented and used as a first bit for next addition step.

The formula for modular multiplication in diminished-1 form of two numbers $a$ and $b$ modulo $2^n + 1$, which is used by this method, is as follows:

$$d(ba) = \left(\sum_{i=1}^{n-1} \oplus b_i d(2^i a) \oplus \overline{Z} \oplus d_1(a)\right) + 1 \bmod (2^n + 1). \tag{22}$$

The first term in parentheses should be done by diminished-1 addition and finally the result should be incremented by one.

A constraint for this method is that it cannot produce a true result if one of the operands was zero. Zero can be detected by testing the most significant bit in diminished-1 representation so there is no need to add an additional hardware module for finding whether an operand is 0.

In this formula $Z$ represents the number of zeros from bit $b_1$ to $b_{n-1}$ and bar represents the one's complement of Z. $d_1(a)$ can be calculated by the following formula:

$$d_1(a) = \overline{b_0} d(a) + b_0 d(2a). \tag{23}$$

$d(2^i a)$ can be computed by the following relation:

$$d(2^i a) = \{a_{n-i-1}a_{n-i-2}\ldots a_0\overline{a_{n-1}a_{n-2}}\ldots\overline{a_{n-i}}\}. \tag{24}$$

In other words, multiplication by $2^i$ is accomplished by a cyclic shift of $i$ bits to the left with the shifted bits being complemented.

An instance of the architecture for this multiplier modulo $2^8 + 1$ is shown in Figure 5.

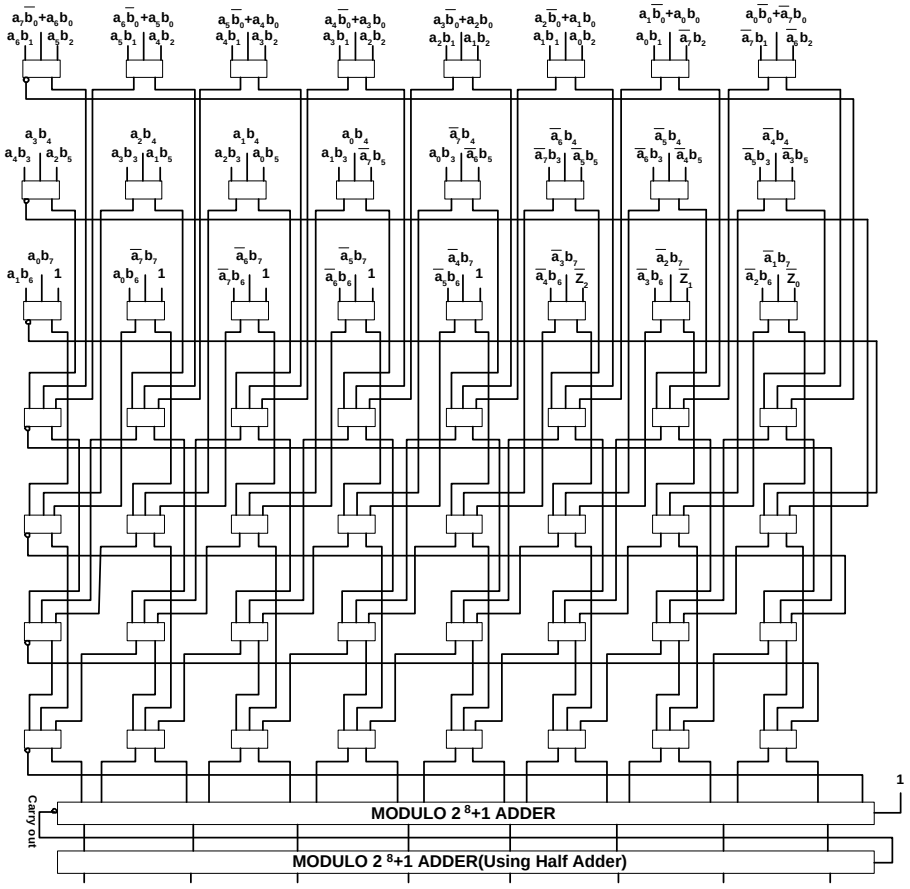



Fig. 5. $2^8 + 1$ modular multiplication architecture

In this figure each rectangle represents a full adder block. As can be seen, due to using Wallace tree all additions are done by CSA architecture and there is no carry propagation in adders.

The additions that are taken in this multiplier are briefly shown in Figure 6.

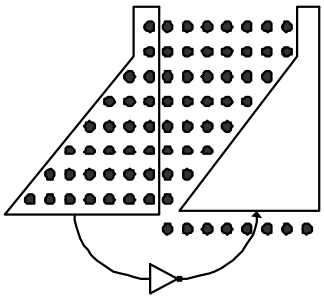


Fig. 6. The additions that are taken in $2^8 + 1$ multiplier

For converting a binary number to diminished-1, one can use the methods that are proposed in [32]. In this paper, the selected methods are shown in the next subsection.

### 10.2 Converting to Diminished-1 in $2^n + 1$

Since $2^n = -1 \bmod (2^n + 1)$, the residue operation is accomplished by $a \bmod (2^n + 1) = a \bmod 2^n - a\, div\, 2^n$.

For calculating the remainder of a number in modulo $2^n + 1$, the quotient of division with $2^n$ (that may be necessary to calculate its remainder in modulo $2^n$) is subtracted from the remainder of this division, in modulo $2^n + 1$. Suppose that 'a' has $k$, $n$-bit blocks; in order to calculate the remainder of modulo $2^n + 1$ these relations are confirmed:

$$a = (a_{k-1}, a_{k-2}, \ldots, a_1, a_0) \tag{25}$$

$$R = a_0 - (a_1 - (a_2 - \ldots - (a_{k-2} - a_{k-1}) \ldots)) \bmod (2^n + 1) \Rightarrow \tag{26}$$

$$R0 = a_0 + a_2 + a_4 + \ldots \bmod (2^n + 1) \tag{27}$$

$$R1 = a_1 + a_3 + a_5 + \ldots \bmod (2^n + 1) \tag{28}$$

$$R = R0 - R1 \bmod (2^n + 1). \tag{29}$$

$R$ can be acquired from $R0$ and $R1$ but calculation of $d(R)$ or $d(R0 - R1)$ is the main goal. Suppose that $x = R0 + 1$ and $y = R1 + 1$ and the relation of diminished-1 is taken; then:

$$\begin{aligned} d(x - y) &= d(x) + \overline{d(y)} + 1 = R0 + \overline{R1} + 1 = d(R0) + 1 + \overline{d(R1) + 1} + 1 \\ &= d(R0) + 1 + 2^n - 1 - d(R1) - 1 + 1 = d(R0) + \overline{d(R1)} + 1 \\ &= d(R0 - R1). \end{aligned}$$

So for calculation of $d(R0 - R1)$ there is no need to calculate each $d(R0)$ and $d(R1)$. Note that in this relation the fact that $\overline{x}$ is one's complement and is equal to $2^n - 1 - x$, is used.

Finally for converting a number to diminished-1 in modulo $2^n + 1$, first one must calculate R0 and R1 by doing diminished-1 addition with the appropriate blocks (note that there is no need to convert these blocks to diminished-1 and just suppose that they are in diminished-1 representation and use diminished-1 addition rule for them).

This is true if the number of blocks in $R0$ and $R1$ were equal because the differences between diminished-1 and binary number are eliminated by subtraction. For instance if $R0$ and $R1$ were not diminished-1 and it is supposed that they are in diminished-1 representation, each of them is more than the desired number. Therefore if each of them has $k$ blocks then it has $k$ values larger than desired and by subtracting $(R0 - R1)$ these extra values are eliminated.

If they are not equal, a zero block must be added. Therefore the final method can be summarized as follows:

1. Calculate the summation of odd and even blocks with diminished-1 rules and CSA architecture.
2. Add the result of summation of even blocks with the one's complement of the result of the odd blocks.
3. Perform full addition in final result that is in CSA format.

Figure 7 shows architecture of 6 blocks. This architecture can be used for both inputs. If there is an input less than 6 blocks all other blocks must be zero.

### 10.3 $2^n - 1$ Modular Multiplier

In this section the architecture of $2^n - 1$ multiplier is proposed by using the architecture of $2^n + 1$ modular multiplication. This architecture is much similar to previous one but is simpler and uses less area. In this implementation Wallace tree is also used.

The property of operations in modulo $2^n - 1$ [14] differ from $2^n + 1$ multiplier as follows:

1. The output carry is used without any changes in the first bit of addition and no complement is necessary.
2. In $2^n - 1$ modular multiplier, there is no need to know the number of zeros from $b_1$ to $b_n - 1$. This block is removed from the architecture and as a result the speed is increased.
3. There is no need to compute $d_1(a)$ and the bits of $b$ are used instead.
4. It is not required to distinguish zero from the other numbers. The output remains correct and produces zero number if each input was zero. So the hardware for zero detection and producing the zero output for this case is removed.

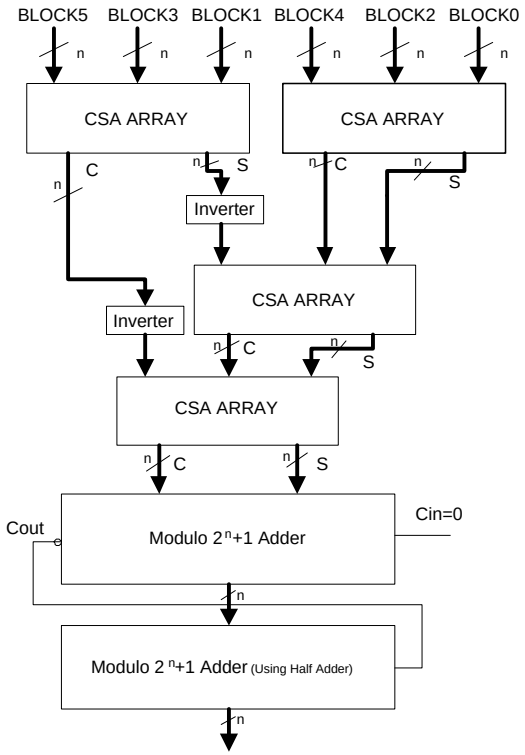



Fig. 7. Architecture for converting to diminished-1 for 6 blocks

With these differences one can discover that this multiplier has more processing speed and less area than $2^n + 1$ multiplier. The architecture of this multiplier modulo $2^8 - 1$ is shown in Figure 8. In this figure there are two inputs – $a$ and $b$. The additions that are done in this multiplier are shown in Figure 9.

For converting to modulo $2^n - 1$ the methods in [33] can be used. The method selected in this paper is summarized in the next subsection.

### 10.4 Converting to Modulo $2^n - 1$

This method is similar to converting to diminished-1 but some changes must be made.

Since $2^n = 1 \bmod (2^n - 1)$, the residue operation is accomplished by

$$a \bmod (2^n - 1) = a \bmod 2^n + a \operatorname{div} 2^n. \tag{30}$$

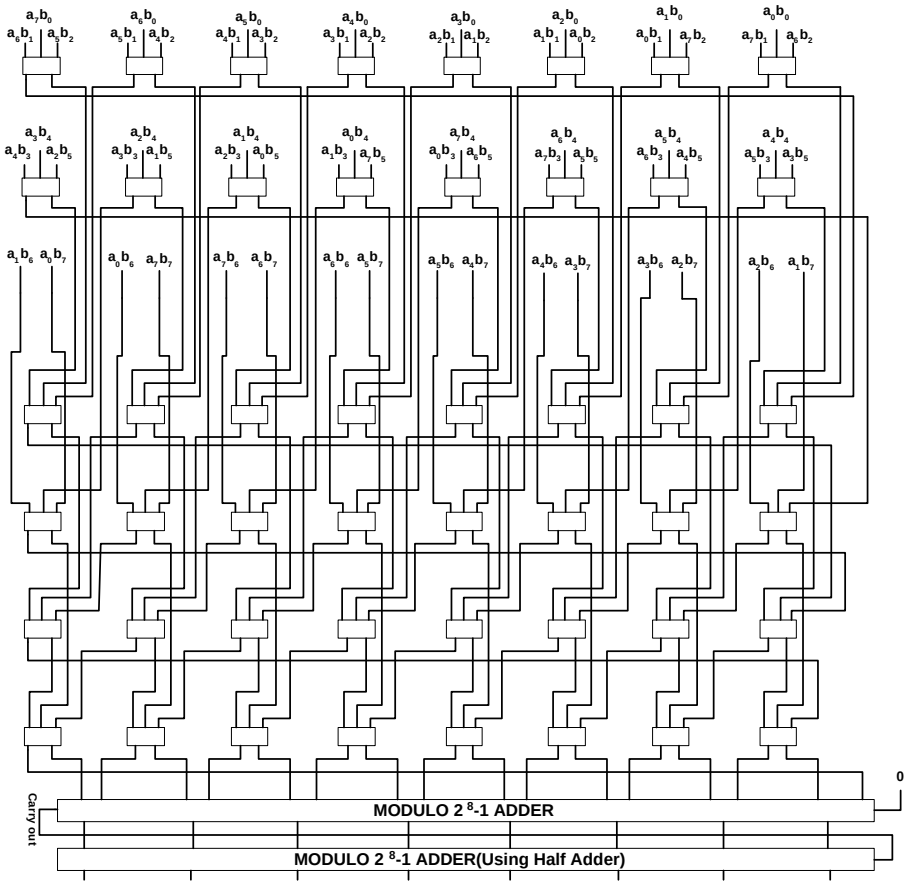



Fig. 8. $2^8 - 1$ modular multiplication architecture

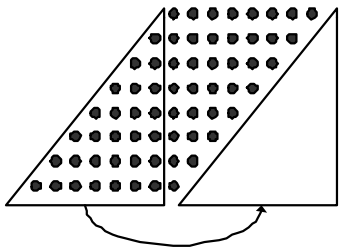


Fig. 9. The additions that are taken in $2^8 - 1$ multiplier

Suppose that 'a' has $k$, $n$-bit blocks; in order to calculate the remainder of modulo $2^n - 1$ the following relations are true:

$$a = (a_{k-1}, a_{k-2}, \ldots, a_1, a_0) \quad (31)$$

$$R = a_0 + a_1 + \ldots + a_{k-1} \bmod (2^n - 1). \quad (32)$$

It is obvious that there is no need to calculate the summation of odd and even blocks separately. The addition is taken in mod $2^n - 1$ and not in diminished-1, so the inverters of previous architecture can be removed. The architecture of this method for 6 blocks is shown in Figure 10.

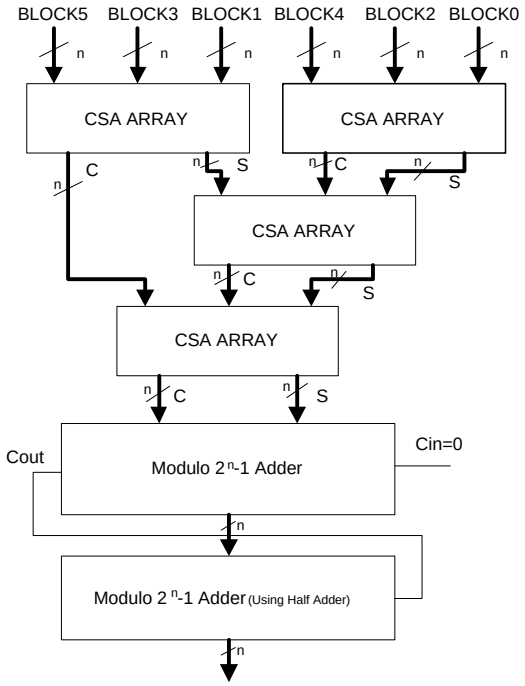



Fig. 10. Architecture for converting to modulo $2^n - 1$ for 6 blocks

### 10.5 $2^n + 1$ Modular Adder

In this section the required architecture for $2^n + 1$ modular adder for RSA implementation in RNS is presented.

To use diminished-1 representation as input of this adder, the input has $n + 1$ bits for a number with $n$ bits. The task of this block is to add the input number with the value of internal register in modulo $2^n + 1$, to save the new result to this register and to send it out.

The internal register is set to zero when resetting this block. Zero in diminished-1 represents the number 1, so the output is incremented by one. This fact changes the number from diminished-1 representation to binary number.

One may think that this representation change makes an incorrect result, but as can be seen, this change can be helpful for hardware implementations because the addition modulo $2^n + 1$ is the last operation of the second base extension and the result of these adders might be used as an input to system or sent out as a result. Also if it is used as input, it goes to modulo $2^n - 1$ multipliers, so in both cases the number should not be in diminished-1 representation.

In this modular adder architecture, number 0 should be discriminated by testing the MSB, while it must not change the output value. This adder block is shown in Figure 11. In this architecture the clock is stopped when a MSB = 1; this is done by a simple circuit that controls the register from loading the new value. Two first additions are used for diminished-1 summation and carry correction.

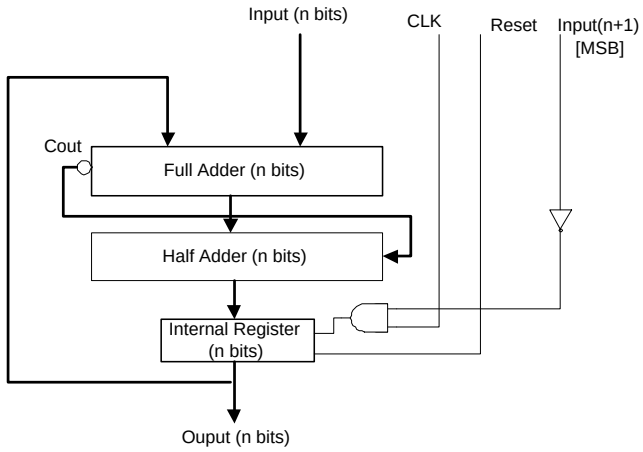



Fig. 11. Modulo $2^n + 1$ adder architecture

### 10.6 $2^n - 1$ Modular Adder

This adder is very similar to the previous one but this architecture does not use diminished-1 representation, so there is no need to do carry correction that should be done in diminished-1 addition by complementing the carry and use it as a first bit of the addition.

A proposed architecture for this modular adder is shown in Figure 12. As shown, there is no need to add a circuit to distinguish zero from other numbers and it can be used like the other numbers.

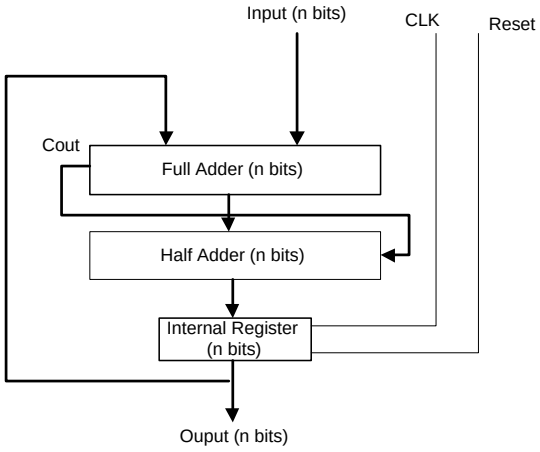



Fig. 12. Modulo $2^n - 1$ adder architecture

## 11 RSA HARDWARE IMPLEMENTATION WITH RNS

Tables 8 and 9 give the number of required multiplications and memories for our improved method and for the Bajard method, respectively.

| | The number of needed multiplication | |
|---|---|---|
| Calculation of the value: | Bajard method | Improved method |
| $\sigma_i$ | 3 | 2 |
| $\xi_j$ | $k+4$ | $k+2$ |
| $\lvert\hat{r}\rvert_{m_r}$ | $k+3$ | $k+2$ |
| $\alpha$ | $k+1$ | $k+1$ |

Table 8. Comparison based on the number of modular multiplications

| | The number of needed memory | |
|---|---|---|
| Calculation of the value: | Bajard method | Improved method |
| $\sigma_i$ | 2 | 1 |
| $\xi_j$ | 4 | 2 |
| $\lvert\hat{r}\rvert_{m_r}$ | 3 | 2 |
| $\alpha$ | 2 | 2 |

Table 9. Comparison based on the number of memory

Note that the number of memory in Table 9 is shown for one modulus; if one multiplies this value by the number of moduli that are used in RNS, the difference between the Bajard method and the improved one is more evident.

The results show that the number of multiplication is reduced by about 33 % in $\sigma_i$ calculations and in other cases this reduction has been varied by the value of $k$ (number of moduli in RNS bases).

In many cases the number of used memories for saving the constant values is reduced by about 50 %. For instance, if the number of moduli was 10, instead of saving 20 constants (2 constants for each modulus), 10 constants should be saved (1 constant for each modulus) in the calculation, i.e., a reduction by 50 %.

The improved method with the discussed processing unit is implemented for RSA 1024 and synthesized with Leonardo Spectrum 2002 for CMOS 0.6 library. The result is shown in Table 10. In this implementation the number of moduli was 10 in each base ($k = 10$).

| Area (Gates) | Clock Speed (MHz) | Throughput Rate (Mb/s) |
|---|---|---|
| 87 8908 | 3.29 | 0.23 |

Table 10. Result of synthesized improved RNS method

This implementation may not fit in one FPGA but if one wants to implement it in a FPGA technology, he/she can use multiple FPGA.

The number of cycles for throughput calculation in the result table is achieved with the following assumptions:

- Cycles for RSA: 14 395
- Cycles to convert the result back to binary form: 18
- Cycles for calculating the result modulo $N$ (if needed): 10

So the total number of cycles is 14 423 and one can use this to calculate the throughput rate.

## 12 COMPARISON BETWEEN TWO RSA IMPLEMENTATIONS

In this section two RSA implementations (RNS and Montgomery) are compared with respect to their used area and processing speed. The Area × Time factor can be used for comparing and selecting the proper method (as used in most papers). One can find that time has straight relation with (1/Throughput), for this reason the factor Area (1/Throughput) or in the general form Area $(1/\text{Throughput})^i$ can be selected for this comparison. The last factor can be used when calculation speed is more important than used area. So no matter how this value was smaller, better performance, less area and more throughput rate result.

For this comparison 1 024 bits version of each implementation is selected. For clarity these values are shown in Table 11. Note that these implementations are done with respect to the architectures that are discussed previously.

The calculated values for the Area (1/Throughput) are as follows: for Montgomery the value is 555 078.6 and for RNS the value is 3 821 339.13. Since the

| | Area (Gates) | Throughput (Mb/s) |
|---|---|---|
| Montgomery | 77 711 | 0.14 |
| RNS | 878 908 | 0.23 |

Table 11. The results of the two methods implementation

values have inverse relation with throughput, the Montgomery method has better performance.

## 13 CONCLUSIONS

This paper presents two major methods for RSA implementation, namely Montgomery and RNS. These two methods are most widely used for RSA hardware implementations. RNS has faster processing speed than Montgomery but due to its parallel architecture, it uses more area for hardware implementation. For comparison two of the best implementations of RNS and Montgomery are selected and modified to enhance their performances.

In this modification, Montgomery is improved by about 45 % in its throughput, has less change in its area and RNS is improved by about 40 % in its used area with some changes in its throughput.

Finally the compared results help select the proper method for implementing the RSA cryptosystem. The result shows that if throughput and the used area have the same significance, Montgomery is the best choice but if throughput is more important then the situation may change.

The results show distinguished differences between RNS and Montgomery implementation. RNS has better performance if throughput has an edge over the area by more than 5 times, but in most common RSA implementations due to limited area and simplicity Montgomery is the favorite choice.

## REFERENCES

[1] Rivest, R. L.—Shamir, A.—Adleman, L.: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of ACM, Vol. 21, 1978, pp. 120–126.

[2] Montgomery, P. L.: Modular Multiplication Without Trial Division. Mathematics of Computation, Vol. 44, 1985, pp. 519–521.

[3] Diffie, W.—Hellman, M. E.: New Directions in Cryptography. IEEE Transactions on Information Theory, Vol. 22, 1976, pp. 644–654.

[4] Bernal, A.—Guyot, A.: Design of A Modular Multiplier Based on Montgomery's Algorithm. In: Proceedings of the 13th Conference on Design of Circuits and Integrated Systems, Spain, November 1998, pp. 680–685.

[5] Blum, T.—Paar, C.: Montgomery Modular Exponentiation on Reconfigurabl Hardware. In: Proceedings of the 14th IEEE Symposium on Computer Arithmetic, 1998, pp. 70–77.

[6] Chao, C.—Chang, T. S.—Jen, C. W.: A New RSA Cryptosystem Hardware Design Based on Montgomery's Algorithm. IEEE Transactions on Circuits and Systems, Vol. 45, 1998, pp. 908–912.

[7] Koc, C. K.—Hung, C. Y.: Carry-Save Adders for Computing the Product AB Modulo N. Electronics Letters, Vol. 26, 1990, pp. 899–900.

[8] Soderstrom, M.: RNS Arithmetic: Modern Application in DSP. IEEE Press, 1986, pp. 418.

[9] Bajard, J. C.—Imbert, L.: A Full RNS Implementation of RSA. IEEE Transactions on Computers, Vol. 53, 2004, pp. 76–774.

[10] Pearson, D.: A Parallel Implementation of RSA. SAC '96, 1996, pp. 1–10.

[11] Posch, K. C.—Posch, R.: Residue Number System: A Key to Parallelism in Public Key Cryptography. IEEE Symposium on Parallel Distributed Processing, 1992, pp. 432–435.

[12] Vu, T. V.: Efficient Implementation of The Chinese Reminder Theorem for Sign Detection and Residue Decoding. IEEE Transaction on Signal Processing, Vol. 47, 1999, pp. 776–783.

[13] Skavantzos, A.—Abdallah, M.: Implementation Issues oftThe Two-Level Residue Number System With Pairs of Conjugate Moduli. IEEE Transactions on Signal Processing, Vol. 47, 1999, pp. 826–838.

[14] Parhami, B.: Computer Arithmetic. Oxford University Press, 2000.

[15] Leibowitz, L. M.: A Simplified Binary Arithmetic for The Fermat Number Transform. IEEE Transaction on Signal Processing, Vol. 24, 1976, pp. 356–359.

[16] Wallace, C. S.: A Suggestion for A Fast Multiplier. IEEE Transactions on Electronic Computers, Vol. 13, 1964, pp. 14–17.

[17] Wang, Z.—Jullien, G. A.—Miller, W. C.: An Efficient Tree Architecture for Modulo 2n+1 Multiplication. Journal of VLSI Signal Processing, Vol. 14, 1996, No. 3, pp. 241–248.

[18] Koc, C. K.—Acar, T.—Burton, S.—Kaliski, J.: Analyzing and Comparing Montgomery Multiplication Algorithms. IEEE Micro, Vol. 16, 1996, pp. 26–33.

[19] Knuth, D. E.: The Art of Computer Programming: Semi Numerical Algorithms. Addison-Wesley, 1981.

[20] McIvor, C.—McLoone, M.—McCanny, J. V.—Daly, A.—Marnane, W.: Fast Montgomery Modular Multiplication and RSA Cryptographic Processor Architectures. Proceedings of the 37th Annual Asilomar Conference on Signals, Systems and Computers, 2003, pp. 379–384.

[21] Elbirt, A. J.—Paar, C.: Towards an FPGA Architecture Optimized for Public-Key Algorithms. SPIE Symposium on Voice, Video and Communications, 1999, pp. 33–42.

[22] Walter, C. D.: Montgomery Exponentiation Needs No Final Subtractions. Electronics Letters, Vol. 35, 1999, pp. 1831–1832.

[23] Guo, J. H.–Wang, C. L.: A Novel Digit-Serial Systolic Array for Modular Multiplication. Proceedings of IEEE International Symposium on Circuits and Systems, ISCAS '98, 1998, pp. 177–180.

[24] Tenca, A. F–Koc, C. K.: A Scalable Architecture for Montgomery Multiplication. CHES '99, 1999, pp. 94–108.

[25] Walter, C. D.: Still Faster Modular Multiplication. Electronics Letters, Vol. 31, 1995, pp. 263–264.

[26] Orup, H.: Simplifying Quotient Determination in High-Radix Modular Multiplication. Proceedings of the 12th Symposium on Computer Arithmetic, 1995, pp. 193–199.

[27] Koc, C. K.: RSA Hardware Implementation. RSA Laboratories, Ver. 1, 1995.

[28] Manochehri, K.—Pourmozafari, S.: Modified Radix-2 Montgomery Modular Multiplication to Make it Faster and Simpler. IEEE Computer Society, Vol. 1, 2005, No. 1, pp. 598–602.

[29] Manochehri, K.—Pourmozafari, S.: Fast Montgomery Modular Multiplication by Pipelined CSA Architecture. ICM'04, Tunisia, 2004, pp. 144–147.

[30] Manochehri, K.—Pourmozafari, S.—Sadeghian, B.: Implementation of RSA Processor with Montgomery Algorithm and CSA Architecture. CSICC2005, Iran, 2005, pp. 703–800.

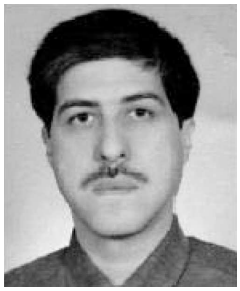
[31] Szabo, N.—Tanaka, R. I.: Residue Arithmetic and Its Application to Computer Technology. MacGraw-Hill, 1967.

[32] Manochehri, K.—Pourmozafari, S.: Studying in Methods of Converting to Diminished-1 in $2n+1$ Modular Multiplication. IKT2005, Iran, 2005, pp. 143–150.

[33] Manochehri, K.—Pourmozafari, S.—Sadeghian, B.: Efficient Methods in Converting to Modulo $2n+1$ and $2n-1$. IEEE Computer Society, 2006, pp. 178–185.

**Kooroush Manochehri** received the B. Sc. degree in computer engineering from the Azad University (Central branch), Iran, in 2001 and the M. Sc. degree in computer engineering (with honors) from the Department of Computer Engineering at the Amirkabir University of Technology, Iran, in 2005. He is a Ph. D. candidate in computer engineering at the Department of Computer EngiNeering of the Amirkabir University of Technology. His research interests are in the fuelds of cryptography, computer arithmetics and hardware implementations.

**Saadat Pourmozafari** is currently an Assistant Professor at the Department of Computer Engineering and IT, Amirkabir University of Technology. He received his M. Sc. degree in microelectronics from ASU (Arizona State University) and his Ph. D. degree from New Mexico State University (New Mexico, USA) in electrical and computer engineering in 1999. His current research interest includes cryptography, computer testing and testable design, VLSI design, fabrication and fault tolerant systems.



**Babak Sadeghian** received his M. Sc. in electrical engineering from Amirkabir University of Technology, Iran, in 1989, and his Ph. D. in computer science from University College, University of New South Wales, Australia, in 1993. His major research interests are in cryptology, especially design and cryptanalysis of block ciphers, and network security, especially intrusion detection systems. He is currently an Associate Professor of computer engineering at IT Department of Amirkabir University of Technology.