Int J Parallel Prog (2009) 37:537–555
DOI 10.1007/s10766-009-0108-7

# High-Performance Hardware of the Sliding-Window Method for Parallel Computation of Modular Exponentiations

**Nadia Nedjah** · **Luiza de Macedo Mourelle**

Received: 16 April 2008 / Accepted: 15 May 2009 / Published online: 11 June 2009
© Springer Science+Business Media, LLC 2009

**Abstract** Modular exponentiation is a basic operation in various applications, such as cryptography. Generally, the performance of this operation has a tremendous impact on the efficiency of the whole application. Therefore, many researchers have devoted special interest to providing smart methods and efficient implementations for modular exponentiation. One of these methods is the sliding-window method, which pre-processes the exponent into *zero* and *non-zero* partitions. *Zero* partitions allow for a reduction of the number of modular multiplications required in the exponentiation process. In this paper, we devise a novel hardware for computing modular exponentiation using the sliding-window method. The partitioning strategy used allows variable-length non-zero partitions, which increases the average number of *zero* partitions and so decreases that of non-zero partitions. It performs the partitioning process in parallel with the pre-computation step of the exponent so no overhead is introduced. The implementation is efficient when compared against related existing hardware implementations.

**Keywords** Modular exponentiation · Partitioning strategy · Modular multiplication · Sliding-window method

N. Nedjah (✉)
Department of Electronics Engineering and Telecommunications, State University of Rio de Janeiro, Rio de Janeiro, Brazil
e-mail: nadia@eng.uerj.br; nadia@pq.cnpq.br

L. de Macedo Mourelle
Department of Systems Engineering and Computation, State University of Rio de Janeiro, Rio de Janeiro, Brazil
e-mail: ldmm@eng.uerj.br

## 1 Introduction

Modular exponentiation is a cornerstone operation in many applications. For instance, it is exploited by several public-key cryptosystems, such as the RSA encryption scheme [15] to encrypt and decrypt information. Modular exponentiation computes $C = T^E \bmod M$, wherein $E$ is called *exponent* and $M$ *modulus*. Here, we assume that $0 \leq T < M$.

Obviously, modular exponentiation consists of a repetition of modular multiplications. Therefore, the performance of the operation is primarily determined by the efficiency of the implementation of the modular multiplication. Consequently, it is of paramount importance to attempt to reduce both the time required to perform a single modular multiplication as well as the number of modular multiplications performed to compute the modular exponentiation.

A simple way to compute $C = T^E \bmod M$ starts by $T \bmod M$ and keeps modular multiplying the partial result over and over again $E$ times. This requires $E - 1$ modular multiplications. This straightforward method computes more multiplications than necessary. For instance, to compute $T^8$, it performs 7 multiplications: $T \rightarrow T^2 \rightarrow T^3 \rightarrow T^4 \rightarrow T^5 \rightarrow T^6 \rightarrow T^7 \rightarrow T^8$. However, $T^8$ can be computed using only 3 multiplications: $T \rightarrow T^2 \rightarrow T^4 \rightarrow T^8$. The answer to the question "what is the fewest number of multiplications to compute $T^E$, given that the only operation allowed is multiplying two already computed powers of $T$?" is an *NP*-complete problem [3,11,12], but there are several efficient algorithms that attempt to reduce the number of multiplications required. Such algorithms are based on a pre-processing of the binary representation of the exponent to group bits into partitions and use repeated square-and-multiply computation to obtain the modular power.

Maybe the most used and simple way to compute modular powers is the *binary* exponentiation method. It uses the binary representation of the exponent ($E = e_{n-1} \ldots e_1 e_0$) as described in Algorithm 1. Note that this is the trivial case as the group of bits is reduced to a single one.

**Algorithm 1** Binary Method ($T$, $M$, $E$)
  $C := T^{e_{n-1}} \bmod M$;
  **For** $i := n - 2$ **downto** 0 **Do**
    $C := C^2 \bmod M$;
    **If** $e_i \neq 0$ **Then** $C := C \times T^{e_i} \bmod M$;
  **Return** $C$;
**End**.

It has been proven that if one partitions the bits of the exponent into larger groups (of more than one bit), one can further reduce the number of required multiplications [2,4,13]. The $m$-ary method uses fixed-length groups of $\log_2 m$, where $m$ is always a power of 2. This method is a generalization of that described in Algorithm 1 and is detailed in Algorithm 2, wherein exponent $E = p_{w-1} p_{w-2} \ldots p_1 p_0$, in which each partition $p_i$ has $d = \log_2 m$ bits and value $v_i$.

**Algorithm 2** $m$-ary Method $(T, M, E)$
  **For** $i := 2$ **to** $2^d - 1$ **Do**
    **Compute** and **Store** $T^i \bmod M$;
  $C := T^{v_{w-1}} \bmod M$;
  **For** $i := w - 2$ **downto** 0 **do**
    $C := C^{2^d} \bmod M$;
    **If** $v_i \neq 0$ **then** $C := C \times T^{v_i} \bmod M$;
  **Return** $C$;
**End**.

Note that for Algorithm 1 and Algorithm 2, the modular multiplication within the loop is only performed when the bit and partition respectively is not zero. So one can attempt to partition the exponent bits such that the *zero* partitions are more frequent and thus the number of modular multiplication in the loop would be reduced. In such a case, however, one has to deal with bit groups of variable lengths. The details of the partitioning strategy and underlying computation will be given in Sect. 2 as it is at the heart of this paper.

The rest of this paper is organized in four sections. First of all, in Sect. 2, we introduce and detail the sliding-window method for modular exponentiation. Subsequently, in Sect. 3, we present the architecture of the proposed hardware implementation. We also give details about the partitioner that performs the exponent pre-processing required by the sliding-window method and we describe a massively parallel implementation of the modular multiplier. Thereafter, in Sect. 4, we assess the performance of the proposed hardware implementation and compare the corresponding characteristics to those of existing related hardware implementations. Last but not least, in Sect. 5, we summarize the content of the paper and draw some useful conclusions.

## 2 Sliding-Window Method

The sliding-window method uses the same logic as the $m$-ary method, except that for the former the window size may vary and hence the exponent partitioning may be performed so that the number of *zero* windows is as large as possible, thus reducing the number of modular multiplications necessary in the multiplication phase [1,2]. Furthermore, as all possible partitions have their most significant bit equal to 1, the pre-computation step needs to be performed for all possible non-zero odd partition values only. This method proceeds as presented in Algorithm 3, wherein exponent $E = p_{w-1}p_{w-2}\ldots p_1p_0$, $\ell_i$ is the length (i.e., number of bits) of partition $p_i$ and as before $v_i$ is the corresponding partition value and $\#\Pi(E)$ is the number of partitions in exponent $E$. Furthermore, $d$ denotes the maximum length of all non-zero partitions. Note that *zero* partitions may be of any possible length.

**Algorithm 3** Sliding Window $(T, M, E)$
  **Parallel begin**
    {*Partitioning phase*}
      **Build** $\Pi(E)$ using the given strategy;
      **Let** $w = \#\Pi(E)$;

{*Pre-computation phase*}
**For** $i := 2, 3$ **to** $2d - 1$ **step** 2 **do**
**Compute** and **Store** $T^i \bmod M$;
**Parallel end**
{*Exponentiation phase*}
$C := T^{v_{w-1}} \bmod M$;
**For** $i := w - 2$ **downto** 0 **Do**
$C := C^{2^{\ell_i}} \bmod M$;
**If** $v_i \neq 0$ **then**
{*Modular multiplication step*}
$C := C \times T^{v_i} \bmod M$;
**Return** $C$;
**End**.

Building $\Pi$ for a given exponent $E$ can proceed using different strategies. For instance, we can use a constant-length non-zero partition strategy or a variable-length non-zero partition strategy [1,2]. The former is described as in Algorithm 4, wherein $d$ is the constant-length of a non-zero partition. The CLNZ partitioning strategy starts a non-zero window when a 1 is encountered. Although the incoming $d-1$ bits may be all 0s, the partitioning algorithm continues to append them to the current window. For instance, if the exponent is $E = 1110000011001$ and $d = 3$ then the windows formed will be $p_3 = 111$, $p_2 = 0000$, $p_1 = 011$ and $p_0 = 001$. The strategy that allows a variable length for non-zero partitions proceeds as described in Algorithm 5, wherein $d$ is the maximum length of a non-zero window and $q$ is the minimum number of zeros required to switch from a non-zero to a *zero* window. The VLNW requires that during the formation of a non-zero window, one should switch to a *zero* partition whenever the remaining bits are all 0s. For the same example of exponent, this strategy would come up with partitions $p_4 = 111$, $p_3 = 00000$, $p_2 = 11$, $p_1 = 00$ and $p_0 = 1$. Note that the VLNZ strategy produces partitions starting and ending with 1s while the CLNZ strategy may produce partition ending with 0, but also always starting with 1. For the above example, we used $q = 2$, $l = 1$ and $r = 0$.

**Algorithm 4** CLNZ $(E, d)$
**ZP:** **Check** the incoming less significant single bit;
**If** it is zero **then Stay** in ZP
**Else Go to** NP;
**NP:** **Stay** in NP until all $d$ bits are collected;
**Check** the incoming single bit;
**If** it is zero **then Go to** ZP
**Else Go to** NP;
**End**.

**Algorithm 5** VLNZ $(E, d, q)$
**ZP:** **Check** the incoming less significant single bit;
**if** it is 0 **then Stay** in ZP
**Else Go to** NP;

**NP:** **Let** $d = l \times q + r + 1$, where $1 < r \leq q$;
**Check** the incoming $q$ bits;
**If** these are all zero **then Go to** ZP
**Else Stay** in NP;
**Stay** in NP until $lq + 1$ bits are received;
At last, the number of incoming bits must be $r$;
**If** these $r$ bits are 0s **then Go to** ZP
**Else Stay** in NP;
After $d$ bits collected, **Check** the incoming bit;
**If** it is zero **then Go to** ZP
**Else Go to** NP;
**End**.

Compared to the *m*-ary method, the sliding-window based on the constant-length non-zero partition strategy increases the number of *zero* windows and so decreases the number of non-zero windows. Therefore, using this strategy we reduce the number of modular multiplications, performed in the multiplication phase (the if-statement in Algorithm 3). The strategy that allows variable-length non-zero windows attempts to increase further the number of *zero* partitions and so reduces further more the number of non-zero partitions. Therefore, the number of multiplications computed in the multiplication phase is also further reduced. The impact of each strategy is depicted in the chart of Fig. 1 (data taken from [1]), depending on the scaled average number of modular multiplications *MN/n* and the total number of bits in the exponent $n$.

In this paper, we focus on the implementation of the modular exponentiation using the sliding method that decomposes the exponent in variable-length non-zero partitions (see Algorithms 3 and 5) in order to reduce to a minimum the number of required modular multiplications. This should yield a faster modular exponentiation hardware when compared to both the one based on the *m*-ary method (see Algorithm 2) and the one based on the constant-length non-zero partitioning strategy (see Algorithms 3 and 4).

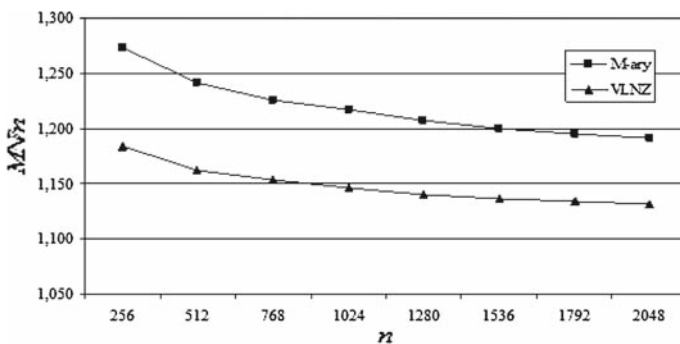


**Fig. 1** Impact of the partitioning strategies

## 3 Proposed Hardware Architecture

The macro-architecture of the hardware for the modular exponentiator is depicted in Fig. 2. Signal *Rst* allows hardware initialization, signal *Start* triggers the commencing of the exponentiation computation, and signal *Final* indicates that the sought modular power (i.e. $T^E \bmod M$) has been obtained and is available in $C$. It uses MODULAR MULTIPLIER that implements the modular multiplication using Montegomery's algorithm (see Sect. 3.3 for details). The exponentiator uses PARTITIONER to take care of the partitioning process as described in Algorithm 5. The output of this process is available in the PARTITION MEMORY, which is controlled by the partitioner. During the exponentiation phase, whenever the exponentiator needs to proceed to the next partition, it asks the partitioner to provide the details of this partition such as type, bit formation in the case of a non-zero window and the length in the case of a *zero* window. In parallel with the partitioning process, the exponentiator computes all the possible modular powers of $T$ for odd exponents, considering the maximum length $d$ of a non-zero partition and stores them in the POWER MEMORY. Later on, in the exponentiation process, it uses these pre-computed powers to yield the expected result. Note that the partial results, obtained during the exponentiation phase, are not stored in the power memory.

The partitioning process applied to exponent for $E$ defined in (1) with $d = 10$ and $q = 4$ produces the data in Table 1, which forms the content of PARTITION MEMORY for $E$. Note that, for these settings, $l = 2$ and $r = 1$ (see Algorithm 5). Besides the type bit, in case of a non-zero partition, the stored word consists of the bits forming the partition and its length. Otherwise, i.e. *zero* partition, the word is $d$ 0s and the length of the partition. The pre-computation step occurs in parallel with the partition step and produces the powers in Table 2. The content of POWER MEMORY is given in the second column of Table 2. As there are $2^{d-1}$ distinct powers to be stored, the address must have $d-1$ bits and is shown in the first column. Note that the address where $T^x$ is stored can be obtained by the $d-1$ most significant bits of $x$.

$$E = \underline{110111011}0000\underline{11}0000\underline{1110111}00\underline{1111110101}0000\underline{11011} \qquad (1)$$

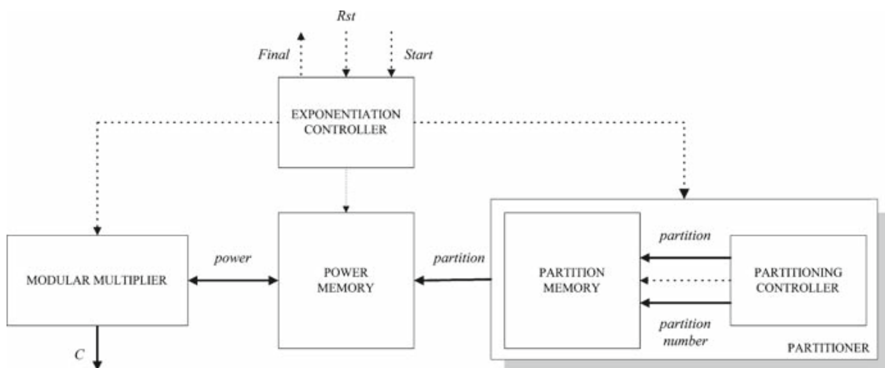


**Fig. 2** Macro-architecture of the sliding-window based exponentiator

**Table 1** Content of PARTITION MEMORY for the example settings

| Type | Formation | Length |
|---|---|---|
| 1 | 0000011011 | 000101 |
| 0 | 0000000000 | 000100 |
| 1 | 1111110101 | 001010 |
| 0 | 0000000000 | 000010 |
| 1 | 0011110111 | 001000 |
| 0 | 0000000000 | 000100 |
| 1 | 0000000011 | 000010 |
| 0 | 0000000000 | 000100 |
| 1 | 0110111011 | 001001 |

**Table 2** Address and content of POWER MEMORY for the example settings

| Address | Power | Current exponent |
|---|---|---|
| 000000000 | $T$ | 0000000001 |
| 000000001 | $T^3$ | 0000000011 |
| 000000010 | $T^5$ | 0000000101 |
| 000000011 | $T^7$ | 0000000111 |
| ... | ... | ... |
| 111111110 | $T^{1021}$ | 1111111101 |
| 111111111 | $T^{1023}$ | 1111111111 |

### 3.1 The Partitioner

Besides the data signals (i.e. $E$, $d$ and $q$) and the result signal *Word*, the interface of the partitioner that uses the strategy described in Algorithm 5 includes three control signals. Signal *pstart* triggers the commencing of the partitioning process and signal *pfinal* indicates that the process has been completed. When *pfinal* $= 1$, the content of the partition memory is ready to be used. The input/output signal *Read* is used, during the exponentiation phase, to start a read cycle of the partition memory. The read word is sent to the exponentiator through signal *word*. As soon as the partitioner answers the requirement of the exponentiator, it withdraws signal $Read$ in preparation for the next read cycle.

The overall hardware architecture of the partitioner is shown in Fig. 3. During the partitioning process, the partition memory is set to work in reading mode. However, as soon as this process is completed, the memory enters the writing mode. The inout signal $Read$ is set by the exponentiator controller when a new read operation is required. The address is computed by decrementing register $Address$ used in the partitioner. Note that when the partitioning process ends, the content of this register is the address of the word containing the details of the last formed partition. The role of the registers used to accomplish the partitioning process is described below. Note that some of these registers are included in Fig. 3 and others show in Fig. 4.

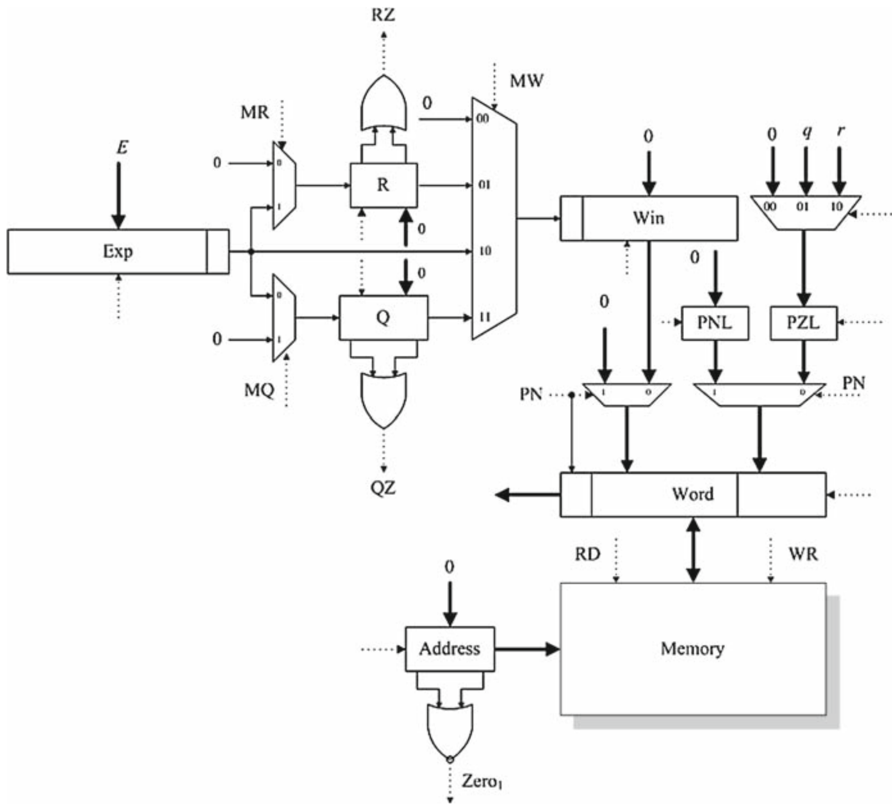


**Fig. 3** Hardware architecture of the partitioner

Register $Exp$ is initialized with the exponent value and is right-shifted when the current least significant bit has been treated. Register $Size$ is initialized with the total number $n$ of bits in the exponent and is decremented whenever register $Exp$ is right-shifted.

Register $Q$ right-shifts the $q$ bits from register $Exp$, after the least significant bit from the actual partition of $Exp$ has been treated. This is done $l$ times, iteratively. For this purpose, register $L$ is first set to $l$ and decremented every time $q$ bits are received from $Exp$. Register $QL$ indicates how many useful bits there are in register $Q$ so far. It is first set to $q$, being decremented every time a right-shift into register $Q$ is done.

Once the $l \times q$ bits have been treated, register $R$ right-shifts $r$ bits from register $Exp$. Similarly to $QL$, register $RL$ indicates how many useful bits there are in register $RL$ so far. It is first set to $r$, being decremented every time a right-shift into register $R$ is done.

Register $Win$ stores the bits of a non-zero partition, in which the least significant bit is right-shifted from register $Exp$, the following $l \times q$ bits from register $Q$ and the final ones $r$ bits from register $R$ to form a partition of $d$ bits at most (according to Algorithm 5). The least and most significant bits of register $Win$ are necessarily 1, which characterizes a non-zero partition.

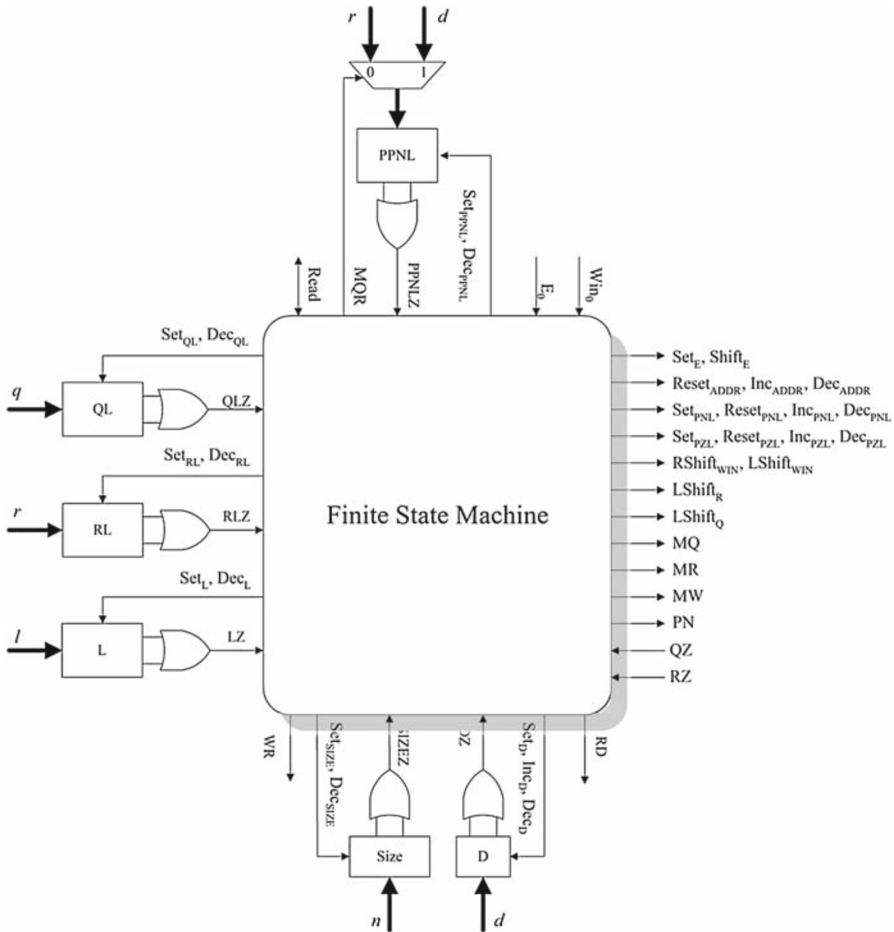


**Fig. 4** Interface signals of the partitioner controller

Register $D$ is used to control the maximum length allowed for a non-zero partition. First, it is set to $d$, then it is decremented whenever register $Win$ is right-shifted and incremented in those occasions when $Win$ is left-shifted as explained later on in this section.

The size of a *zero* partition is stored in register PZL, while that a non-zero partition is kept in register PNL. Once in a *zero* partition and every time a 0 is received from $Exp$, PZL is incremented. Note that the bits of a *zero* partition are never registered. On the other hand, once in a non-zero partition and every time register $Win$ is right-shifted, PNL is incremented, except during the padding process, as explained later on in this section. The switching from a non-zero to a *zero* partition occurs in one of the following situations:

- The $q$ bits in register $Q$ are all 0s: this means that we must switch to a *zero* partition. So, register $Win$ must be padded with 0s as to obtain the right value of the current

non-zero partition. The 0s in register $Q$ are the first bits of the new *zero* partition. Therefore, register PZL must be set to $q$, in order to account for these bits.
- The $r$ bits in register $R$ are all 0s: this also means that we must end the current non-zero partition and switch to a *zero* window. As in the previous case, register $Win$ must be padded with 0s. However, in this case, register PZL must start with $r$ to account for the 0s that are in register $R$.

In both cases, before the padding occurs, if the most significant bits of $Win$ are 0s, these must be accounted for in the new *zero* partition. This is guaranteed by left-shifting $Win$, incrementing registers PZL and $D$ and decrementing PNL.

For the last partition, which must be a non-zero one, the bits shifted out from $Exp$ may stop at any time before completing the $d$ bits. This can happen in one of the following situations:

- The bits of $Exp$ ended before completing the $q$ bits to fill in register $Q$: register PPNL must be set to $q$. Besides, register $Q$ must be left-padded with a number of 0s as dictated by the content of register QL. Each time a 0 is shifted into $Q$, register PPNL is decremented. At the end, the value registered in PPNL indicates the exact number of right-shifts required from $Q$ to $Win$.
- The bits of $Exp$ ended before completing $r$ bits to fill in register $R$: register PPNL, in this case, must be set to $r$. Furthermore, register $R$ must be left-padded with a number of 0s as indicated by the content of register RL. As before, each time a 0 is shifted into $R$, register PPNL is decremented. At the end, the value registered in PPNL indicates the exact number of right-shifts required from $R$ to $Win$.

Every time the starting bit of a new partition is identified, the data of the current one are stored in the partition memory. These data are:

- The type of the partition, indicated by signal PN;
- Either the bit formation in register $Win$ or $00 \cdots 0$, depending on whether PN is 1 or 0, respectively;
- The length of the partition, which is kept in register PZL or PNL, again depending on whether PN is 0 or 1, respectively.

The controller of the partitioning process is implemented a synchronous finite state machine of 37 states. State transitions occur as described as follows. Note that when the current state is $S_i$ and no next state is explicitly specified, the transition to next state $S_{i+1}$ occurs. Note that PMEM represents the PARTITION MEMORY. The interface of the controller of the partitioner is shown in Fig. 4.

```
S0:  Load E; Set Size; Reset Address; Reset PZL; RD := 1; WR := 0;
     If pStart = 0 Then Go to S0 Else Go to S1;
S1:  If Size = 0 Then Go to S34;
     Else If E0 = 0 Then PN := 0; Go to S2;
          Else PN := 1; Decrement Size; Go to S5;
S2:  Reset Win; Set D; Left-Shift E; Increment PZL; Decrement Size;
S3:  If E0 = 0 and Size /= 0 Then Go to S2
     Else PMEM [Address].Type := PN; PMEM [Address].Length := PZL;
          Increment Address; Reset PZL; Go to S4;
S4:  If E0 = 1 Then PN := 1; MW := 2; Go to S5 Else Go to S0;
```

```
S5:  Reset PNL; Set D; Reset Q; Set L; Set QL; MQ := 0;
S6:  Left-Shift Q; Decrement QL;
S7:  Left-Shift E; Decrement Size;
S8:  If Size /= 0 Then
        If QL /= 0 Then Go to S6;
        Else If QZ = 0 Then MW := 0; Go to S27
             Else MW := 3; Go to S9;
     Else MQ := 1; Set PPNL with q; Go to S20;
S9:  Set QL;
S10: Left-Shift Win; Increment PNL; Decrement D; Decrement QL;
S11: If QL = 0 Then Decrement L; Go to S12
     Else Left-Shift Q; Go to S10;
S12: If L = 0 Then Set RL; MR := 0; Go to S14;
     Else Set QL; Go to S6;
S13: Left-Shift E; Decrement Size;
S14: Left-Shift R; Decrement RL;
S15: If Size /= 0 Then
        If RL /= 0 Then Go to S13;
        Else If RZ = 0 Then Set PZL with r; Go to S32;
             Else MW := 1; Go to S16;
     Else MR := 1; Set PPNL with r; Go to S24;
S16: Set RL;
S17: Left-Shift Win; Increment PNL; Decrement D; Decrement RL;
S18: If RL = 0 Then Go to S19 Else Shift R; Go to S17;
S19: PMEM[Address].Type := PN; PMEM[Address].Length := PNL;
     PMEM[Address].value := Win; Increment Address; Shift E;
     Decrement Size; Reset PZL; Go to S1;
S20: Left-Shift Q; Decrement QL; Decrement PPNL;
S21: If QL = 0 Then Set QL; MW := 3; Go to S22 Else Go to S20;
S22: Left-Shift Win; Increment PNL; Decrement D; Decrement PPNL;
S23: If PPNL = 0 Then MW := 0; Go to S28 Else Go to S22;
S24: Left-Shift R; Decrement RL; Decrement PPNL;
S25: If RL = 0 Then Set RL; MW := 1; Go to S26 Else Go to 21;
S26: Left-Shift Win; Increment PNL; Decrement D; Decrement PPNL;
S27: If PPNL = 0 Then MW := 0; Set PZL with q; Go to S28;
     Else Go to S26;
S28: Left-Shift Win; Decrement D;
S29: If D = 0 Then Go to S30 Else Go to S28;
S30: PMEM[Address].Type := PN; PMEM[Address].Length := PNL;
     PMEM[Address].Value := Win; Increment Address;
S31: If Size /= 0 Then PN := 0; Go to S2 Else Go to S34;
S32: If Win[d-1] = 0 Then Go to S33 else MW := 0; Go to S28;
S33: Right-Shift Win; Decrement PNL; Increment D;
     Increment PZL; Go to S32;
S34: pFinal := 1; WR := 1; RD := 0;
S35: If Read = 1 Then Go to S36;
     Else If pStart = 0 Go to S0 Else Go to 35;
S36: Decrement Address; Go to S35;
```

### 3.2 The Modular Multiplier

One of the widely used algorithms for efficient modular multiplication is Montgomery's algorithm [5,8,9],, This algorithm computes the modular product of two integers *modulo* a third one without performing any trial divisions. It yields the reduced product using a series of additions. Let $A$, $B$ and $M$ be the operands such that we would like

to compute $R = A \times B \bmod M$. The pre-conditions for the application of Montgomery's algorithm are as follows: $(i)$ the modulus $M$ needs to be relatively prime to the radix, i.e. there exists no common divisor for $M$ and the radix; $(ii)$ the multiplicand and the multiplier need to be smaller than $M$. As we use the binary representation of the operands, then the modulus $M$ needs to be odd to satisfy the first pre-condition. The Montgomery algorithm uses the least significant digit of the accumulating modular partial product to determine the multiple of $M$ to subtract. The usual multiplication order is reversed by choosing multiplier digits from least to most significant and shifting down. If $R$ is the current modular partial product, then $q$ is chosen so that $R + q \times M$ is a multiple of the radix $r$, and this is right-shifted by one position, i.e. divided by $r$ for use in the next iteration. Consequently, after n iterations, the result obtained is $R = A \times B \times r^{-n} \bmod M$. In order to yield the exact result, we need an extra Montgomery modular multiplication $(2^n \bmod M) \times R \bmod M$. The Montgomery algorithm is given in Algorithm 6.

**Algorithm 6** Montgomery $(A, B, M)$
$R := 0;$
**For** $i := 0$ **to** $n - 1$ **Do**
$R := R + A[i] \times B;$
**If** $R \bmod 2 = 0$ **then** $R := R \operatorname{div} 2$
**Else** $R := (R + M) \operatorname{div} 2;$
**Return** $R$;
**End**.

A modified version of Montgomery algorithm is given in Algorithm 7. The least significant bit of $R + a_i \times B$ is the least significant bit of the sum of the least significant bits of $R$ and $B$ if $a_i$ is 1 and the least significant bit of $R$ otherwise. Furthermore, new values of $R$ are either the old ones summed up with $a_i \times B$ or with $a_i \times B + q_i \times M$ depending on whether $q_i$ is 0 or 1, wherein $Q$ represents the quotient $(A \times B)/M$.

**Algorithm 7** ModifiedMontgomery (*A, B, M*)
$R := 0;$
**For** $i := 0$ **to** $n - 1$ **Do**
$Q[i] := (R[0] + A[i] \times B[0]) \bmod 2;$
$R := (R + A[i] \times B + Q[i] \times M) \operatorname{div} 2;$
**Return** $R$;
**End**.

Consider the expression $R + a_i \times B + q \times M$ in Algorithm 7. It can be computed as indicated in the last column of the Table 1 depending on the value of the bits $a_i$ and $q$. In Table 3, *MB* represents the sum $M + B$. A bit-wise version of Algorithm 7 that is at the basis of our systolic implementation, is described in Algorithm 8.

**Algorithm 8** SystolicMontgomery $(A, B, M, MB)$
$R := 0;\ R_1 := 0;\ carry := 0;\ t := 0;$
**For** $i := 0$ **to** $n$ **Do**

**Table 3** Computation of $R + a_i \times B + q_i \times M$

| $a_i$ | $q_i$ | $R + a_i \times B + q_i \times M$ |
|---|---|---|
| 1 | 1 | $R + MB$ |
| 1 | 0 | $R + B$ |
| 0 | 1 | $R + M$ |
| 0 | 0 | $R$ |

$Q[i] := R[0]$ xor $A[i]$ and $B[0]$;
**For** $j := 0$ to $n$ **Do**
**Switch** $A[i]$, $Q[i]$
1,1: $t := MB[i]$;
1,0: $t := B[i]$;
0,1: $t := M[i]$;
0,0: $t := 0$;
$R[j] := R_1[j+1]$ xor $t$ xor $carry$;
$carry := R_1[j+1]$ and $t$ or $R_1[j+1]$ and $carry$ or $t$ and $carry$;
$R_1 := R$;
**Return** $R$;
**End**.

All algorithms, i.e. Algorithms 6, 7 and 8 are equivalent. They yield the same result. In Algorithm 8, $MB$ represents the result of $M + B$, which at most has $n + 1$ bits. Notice that $R_1$ is $R$ in the previous iteration.

Assuming the Algorithm 8 as basis, the main processing element *PE* of the systolic architecture of the Montgomery modular multiplier computes a bit $r_j$ of residue $R$. This represents the computation of line 8. The left border PEs of the systolic arrays perform the same computation but beside that, they have to compute bit $q_i$ as well. This is related to the computation of line 1. The duplication of the PEs in a systolic form implements the iteration of line 0. The systolic architecture of the modular Montgomery multiplier is shown in Fig. 5.

The architecture of the basic PE, i.e. $cell_{i,j}$, $1 \leq i \leq n-1$ and $1 \leq j \leq n-1$, is shown in Fig. 6a. It implements the instructions of lines 2–9 in systolic Montgomery algorithm of Algorithm 8. The architecture of the right-most top-most PE, i.e. $cell_{0,0}$, is given in Fig. 6b. Besides the computation of lines 2–9, it implements the computation indicated in line 1. However as $r_0^{(0)}$ (i.e., least significant bit of residue in iteration 0) is zero, the computation of $q_0$ is reduced to $a_0.b_0$. Besides, the full-adder is not necessary as carry in signal is also 0 so $r_1^{(0)}$ xor $t$ xor $carry$ and $r_1^{(0)}.t + r_1^{(0)}.carry + t.carry$ are reduced to $t$ and 0 respectively. The architecture of the rest of the PEs of the first column is shown in Fig. 6c. It computes $q_0$ in the more general case, i.e. when $r_0^{(i)}$ is not null. Moreover, the full-adder is substituted by a half-adder as the carry in signals are zero for these PEs. The architecture of the left border PEs, i.e. $cell_{0,j}$, is given in Fig. 6d. As $r_n^i = 0$, the full-adder is unnecessary and so it is substituted by a half-adder.

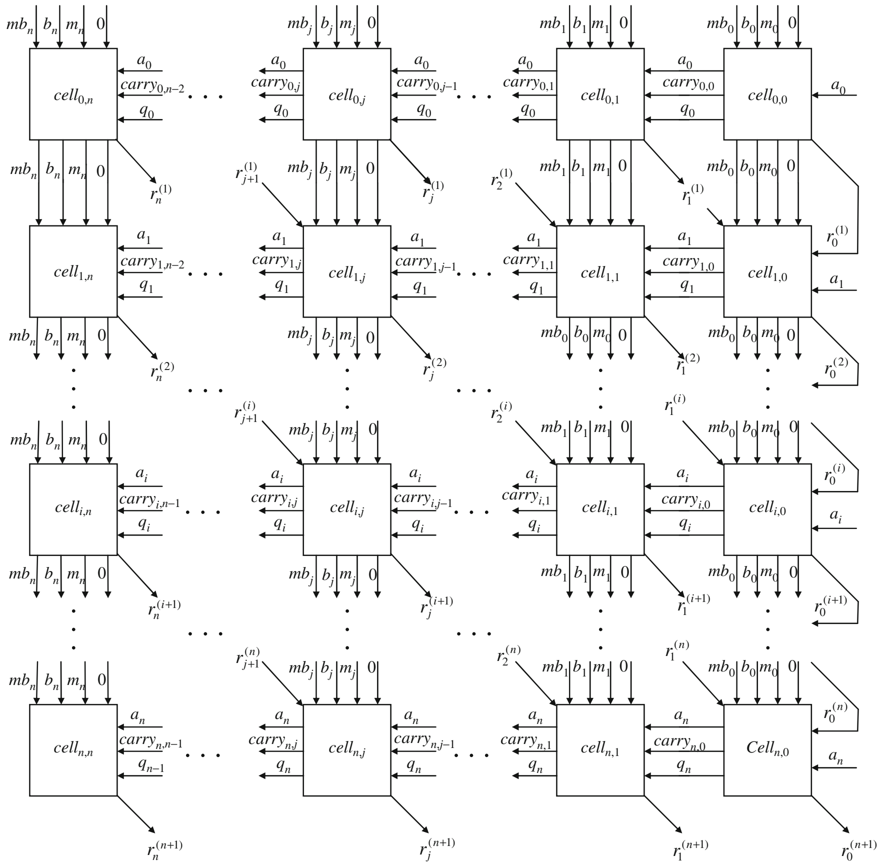


**Fig. 5** Systolic architecture of Montgomery modular multiplier

## 3.3 The Modular Exponentiator

As explained in Sect. 2, the data signals used by the exponentiator are $T$, $M$, $E$, $n$, $d$ and $q$. They are part of the interface of the hardware that implements the sliding-window methods (see Algorithms 3 and 5). Recall that signal $n$ provides the total number of bits in exponent $E$, signal $d$ provides the maximum number of bits allowed for a non-zero window and signal $q$ provides the minimum number of 0s to switch from a non-zero to a *zero* partition.

Component MERSENNE, in Fig. 7, is used to generate the constant $2^d - 1$, which defines the maximum exponent to be pre-computed. Component COUNTER1 is used to address the power memory. During pre-computation, it stores the current odd exponent and is incremented by two for every new operation. Excluding its least significant bit, COUNTER1 provides the address where to store the computed power (see Table 2, in Sect. 3). In order to terminate the pre-computation step, the contents of COUNTER1 is compared to the constant generated by MERSENNE. During the exponentiation step, COUNTER1 is loaded with the current partition value fetched from the partition

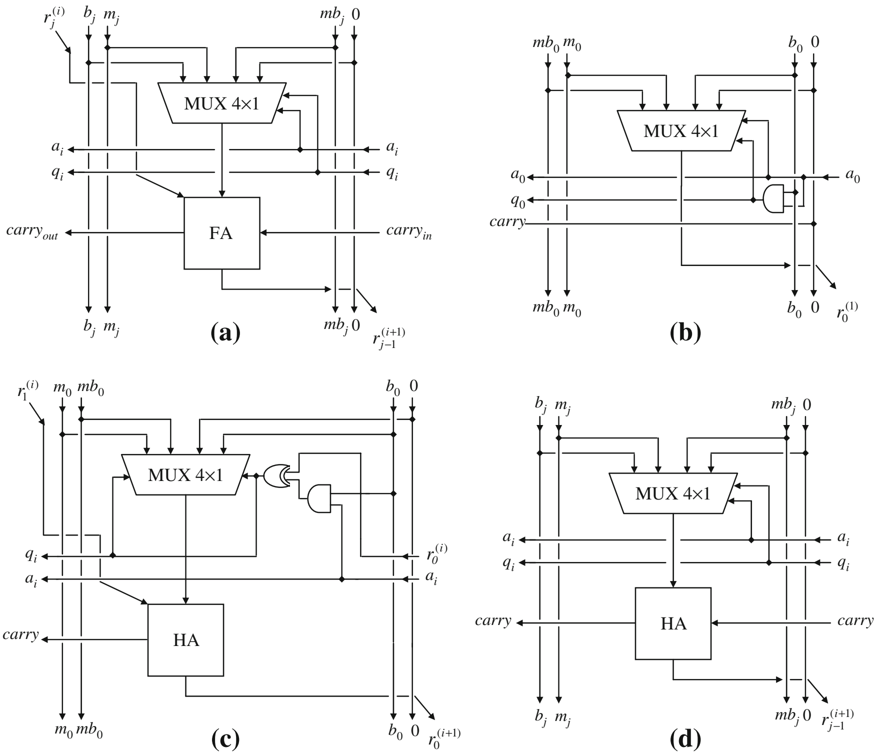


**Fig. 6** Processing elements of the modular multiplier. **a** Basic PE architecture; **b** Right-most top-most PE $cell_{0,0}$; **c** Right border PEs $cell_{i,0}$; **d** Left border PEs $cell_{0,j}$

memory, which allows to address the power memory in the same manner as in the pre-computation step.

Component COUNTER2 initially stores the current partition length, which determines the number of required squaring and is decremented every step of this operation. The proposed architecture of the exponentiator is shown in Fig. 7.

As for the partitioner, the work of the components constituting the modular exponentiator is synchronized using a finite state machine of 21 states. The transitions between these states along with the corresponding output actions are described as follows, wherein $R$ references the partial modular product computed at that point. Observe that as soon as the pre-computation phase starts, it triggers the partitioning phase (see state S1). These two steps proceed in parallel. Once the pre-computation has been completed the controller waits for the partitioning work to end (see state S8), so it can carry on with the exponentiation phase to yield the expected result. Recall that the most significant bit of the partition memory word indicates the current partition type (see Table 1) and is used by the exponentation controller to decide whether the multiplication step is required. The power memory is referenced as WMEM

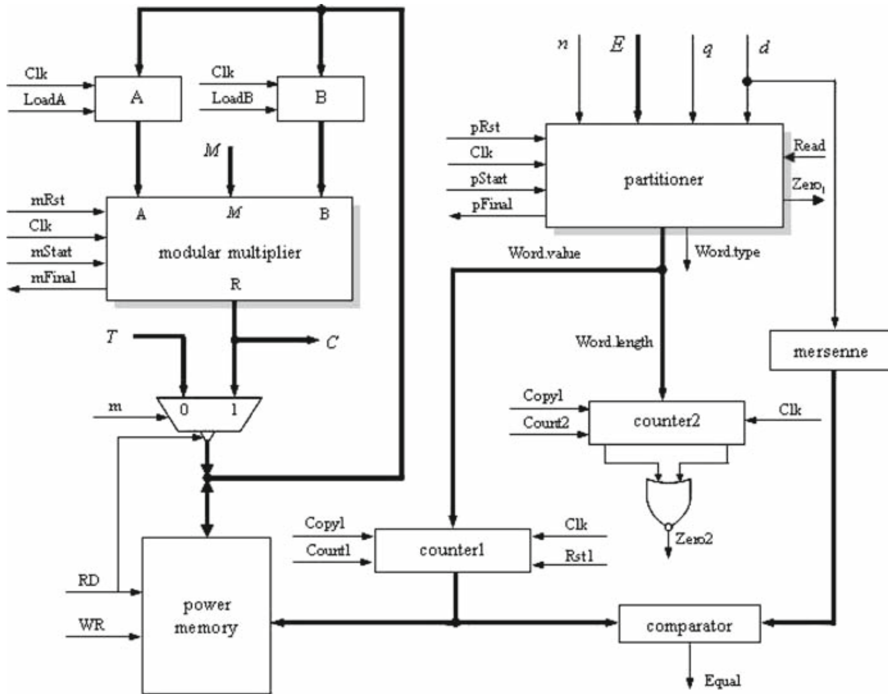


**Fig. 7** Detailed architecture of the sliding-window based exponentiator

```
S0:  If Start = 1 Then Go to S1;
S1:  pStart := 1; Reset COUNTER1; m := 0; WR := 1;
S2:  WMEM[COUNTER1] := T; Load A; Load B;
S3:  mstart := 1; If mFinal = 1 Then m := 1; Go to S4;
S4:  Load R into B;
S5:  mstart := 1; If mFinal = 1 Then Go to S6;
S6:  Increment COUNTER1;
S7:  WMEM[COUNTER1] := R; Load A;
     If Equal = 1 Then Go to S5 Else Go to S8;
S8:  If pFinal = 1 Then Go to S9;
S9:  Read := 1;
S10: Set COUNTER1 with Word.value;
     Set COUNTER2 with Word.length;
S11: Read := 0; RD := 1;
     If Zero2 = 0 Then Go to S12 Else Go to S18;
S12: Load A; Load B;
     If Zero1 = 0 Then Go to S13 Else Go to S20;
S13: Read := 1;
S14: Set COUNTER1 with Word.value;
     Set COUNTER2 with Word.length;
S15: mstart := 1;
     If mFinal = 1 and Zero2 = 0 Then m := 1; Go to S16;
     Else If mFinal = 1 and Zero2 = 1 Then Go to S19;
S16: Load A; Load B; Decrement COUNTER2;
S17: If Zero2 = 1 and Word.type = 1 Then Go to S11;
     Else If Zero2 = 1 and Word.type = 0 and Zero1 = 0 Then
             Go to S13;
```

```
     Else If Zero2 = 1 and NZW = 0 and Zero1 = 1 Then
             Go to S20;
S18: Load B; Go to S15;
S19: Load A; Load B;
     If Zero1 = 1 Then Go to S20 Else Go to S13;
S20: final := 1; If start = 0 Then Go to S0;
```

## 4 Performance Results

We modeled the proposed architecture for the sliding-window method with the variable-length non-zero window partitioning strategy for processing modular exponentiation using VHDL [6]. Then, we functionally simulated the obtained specification and implemented it through automatic synthesis in reconfigurable FPGAs of the Spartan family. The reported area (in CLBs) and time (in ns) requirements are given in Table 4. Unfortunately, it was impossible for us to obtain the power consumption of the design as this characteristics is not provided in the synthesis report. We compare these figures to those required by the hardware implementation based on *m*-ary method [7,16] and that of the sliding-window with the constant-length non-zero window partitioning strategy [14]. In Table 4, $n$ is the number of bits in the exponent, $d$ is $\log_2 m$ for the *m*-ary method and it represents the constant-length of non-zero partitions in the CLNZ sliding-window and the maximum bits in a non-zero partition in the case of VLNZ sliding-window. Note that for VLNZ sliding-window, the minimum number of 0s before *zero* partition used, i.e. $q$ is 2 for all listed cases.

From the performance results in Table 4, it can be observed that despite the hardware area required by the proposed exponentiator, the response time was considerably reduced and so was the performance factor $1/(area \times time)$. This effect is

**Table 4** Area and time requirements for the VLNZ versus CLNZ sliding-window and *m*-ary based hardware implementations

| $n$ | $d$ | *M*-ary Hard. | | CLNZ Hard. | | VLNZ Hard. | |
|---|---|---|---|---|---|---|---|
| | | Area | Time | Area | Time | Area | Time |
| 64 | 3 | 392 | 16.1 | 598 | 14.2 | 567 | 12.3 |
| | 4 | 404 | 15.4 | 606 | 13.7 | 678 | 10.4 |
| | 5 | 683 | 17.3 | 717 | 13.5 | 815 | 9.2 |
| 128 | 3 | 811 | 22.0 | 723 | 18.5 | 899 | 13.7 |
| | 4 | 821 | 20.1 | 872 | 14.2 | 992 | 10.1 |
| | 5 | 1002 | 24.5 | 1178 | 13.6 | 1355 | 7.3 |
| 256 | 3 | 1351 | 39.8 | 1565 | 26.3 | 1793 | 19.2 |
| | 4 | 1405 | 38.3 | 1690 | 22.3 | 1900 | 15.0 |
| | 5 | 1688 | 44.4 | 1778 | 20.9 | 2114 | 8.5 |
| 512 | 3 | 2229 | 65.4 | 2809 | 39.2 | 2910 | 26.3 |
| | 4 | 2385 | 62.9 | 3079 | 31.9 | 3212 | 18.9 |
| | 5 | 2661 | 69.1 | 3102 | 27.2 | 3378 | 12.3 |

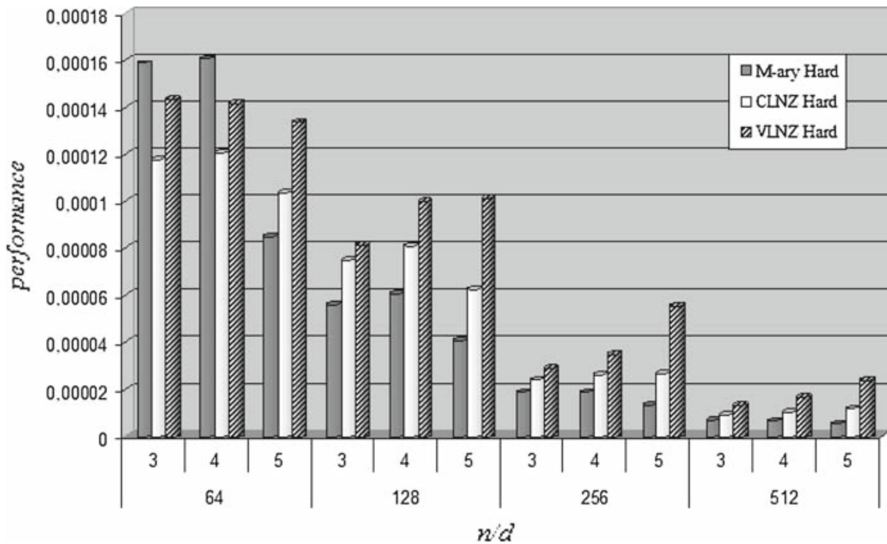


**Fig. 8** Comparison of the performance factor for the implementations based on *m*-ary, CLNZ and VLNZ sliding-window

graphically shown in Fig. 8, wherein the experiments are numbered sequentially. For each of the experiments, the same exponents were used with the three implementations. The improvement of this implementation are mainly due to the maximization of the number of *zero* partitions, which require no multiplication step, reducing thus the total number of executed multiplications.

## 5 Conclusion

In this paper, we propose a novel hardware architecture for the modular exponentiation method based on the sliding-window method. The exponent partitioning strategy used allows variable-length non-zero partitions as well as variable-length *zero* partitions. Compared with related work, the engineered implementation presents a very high throughput, which is necessary in cryptographic systems. It minimizes the number of required multiplications. Besides, it performs the partitioning process in parallel with the pre-computation step of the exponent so no overhead is introduced. The performance of the proposed implementation is further enhanced with the use of a massively parallel implementation of the modular multiplication.

The architecture was specified in VHDL and the obtained specification was simulated and its functionality validated. The hardware specification was synthesized so as to obtain some figures about the hardware area required as well as the time delay imposed by the implementation. The figures of merit were compared to those obtained with related hardware implementations of the modular exponentiation. The first implementation used for performance comparison is based on the *m*-ary method while the second is based on the sliding-window method but with the constant-length

non-zero partition strategy. Of course, in the latter implementation, zero-windows are also allowed to be of variable length. The comparison allowed us to assess the quality and practicality of this hardware. For all the cases we considered, the performance factor, which is the product of the requirements of hardware area and signal propagation delay for the proposed exponentiator is much smaller than that obtained for the other two implementations.

In the future, we intend to investigate the co-design methodology [10] that allows us to implement the partitioning process in software while keeping the modular multiplication in hardware. This solution should reduce to a minimum the hardware resources required and hopefully, yield a good response time, as proven through the performance factor of the three designs that is depicted in the chart of Fig. 8 in the previous section.

**Acknowledgments** We are grateful to CNPq, The National Council for Scientific and Technological Development of the Brazilian Federal Government and FAPERJ, The Research Support Foundation of Rio de Janeiro of the State Government of Rio de Janeiro for their continuous financial support.

## References

1. Koç, Ç. K.: High-speed rsa Implementation. Technical Report. RSA Laboratories, Redwood City (1994)
2. Knuth, D.: The Art of Programming: Semi-numerical Algorithms, vol. 2, 2nd edn. Addison-Wesley, Reading (1981)
3. Kunihiro, N., Yamamoto, H.: New methods for generating short addition chain. IEICE Trans. **E83-A**(1), 60–67 (2000)
4. Menezes, A., Oorschot, P.V., Vanstone, S.: Handbook of Applied Cryptography. CRC Press, USA (1996)
5. Montgomery, P.L.: Modular multiplication without trial division. Math. Comput. **44**, 519–521 (1985)
6. Navabi, Z: VHDL—Analysis and Modeling of Digital Systems, 2nd edn. McGraw Hill, USA (1998)
7. Nedjah, N., Mourelle, L.M.: Four hardware implementations for the *m*-ary modular exponentiation. In: Proceedings of 3rd International Conference on Information Technology: New Generations, pp. 210–215. IEEE Computer Society, Las Vegas (2006)
8. Nedjah, N., Mourelle, L.M.: Hardware architecture for booth-barrett's modular multiplications. Int. J. Model. Simul. **26**(3), 1–8 (2006)
9. Nedjah, N., Mourelle, L.M.: Three hardware architectures for the binary modular exponentiation: sequential, parallel and systolic. IEEE Trans. Circuits Syst. I: Fundam. Theory Appl. **53**(3), 627–633 (2006)
10. Nedjah, N., Mourelle, L.M.: Co-design for System Acceleration: A Quantitative Approach. Springer (former Kluwer Academics), Netherlands (2007)
11. Nedjah, N., Mourelle, L.M.: Efficient and secure cryptographic systems based on addition chains: hardware design vs. software/hardware co-design. Integration, the VLSI J. Elsevier **40**(1), 36–44 (2007)
12. Nedjah, N., Mourelle, L.M.: Fast hardware for modular exponentiation with efficient exponent preprocessing. J. Syst. Archit. **53**(2–3), 99–108 (2007)
13. Nedjah, N., Mourelle, L.M.: Parallel computation of modular exponentiation for fast cryptography. Int. J. High Perform. Syst. Archit. **1**(1), 44–49 (2007)
14. Nedjah, N., Mourelle, L.M., da Silva, R.: Efficient hardware for modular exponentiation using the sliding-window method. In: Proceedigs of 4th International Conference on Information Technology: New Generations, pp. 17–24. IEEE Computer Society, Las Vegas (2007)
15. Rivest, R., Shamir, A., Adleman, L.: A method for obtaining digital signature and public-key cryptosystems. Commun. ACM **21**, 120–126 (1978)
16. Silva, R.M., Nedjah, N., Mourelle, L.M.: Efficient hardware for modular exponentiation using the sliding-window method. Int. J. High Perform. Syst. Archit. **1**(3), 199–206 (2008)