ISSN 2007-9737

Efficiency Evaluation of a Modified Montgomery Multiplication Systolic Architecture Implemented on an FPGA

José de Jesús Morales-Romero[1,*], Mario Alfredo Reyes-Barranca[1], David Tinoco-Varela[2], Luis Martín Flores-Nava[1], Emilio Rafael Espinosa-García[1]

[1] Centro de Investigación y de Estudios Avanzados, Ciudad de Mexico, Mexico

[2] Universidad Autónoma de México, Facultad de Estudios Superiores Cuautitlán, Mexico

{jmoralesr, mreyes, lmflores, espinosa}@cinvestav.mx, dativa19@hotmail.com

Abstract. This work presents an improved algorithm applied to a systolic architecture when a modular multiplication is synthesized into a Field Programmable Gate Array (FPGA). Here, we proved how this proposed architecture for modular multiplication can be employed in a modular exponentiation process. Modular exponentiation is critical and helps in the performance of algorithms like RSA, Digital Signature, Elliptic Curve, and other cryptographic algorithms. Results obtained show that these improvements in the systolic architecture speed up the performance and reduces also the resources used by the programmable device, specifically when the Montgomery modular multiplication is used. Also, we compare the results of this work with related work published in the literature.

Keywords. Modular multiplication, montgomery modular multiplication, modular exponentiation, systolic architecture, FPGA, RSA, elliptic curve cryptography.

1 Introduction

It is well known that cryptographic algorithms like Elliptic Curve Cryptography (ECC), and ElGamal use modular multiplication; especially the modular exponentiation uses modular multiplication. For instance, among other algorithms, RSA is the most used, since proposed by Rivest, Shamir, and Adleman in the year 1977 at the MIT [16].

RSA is based on the modular exponentiation to encrypt and decrypt critical data. Regarding this mentioned algorithm, it can be said that the main operation used is modular multiplication.

Even so, some difficulties are dealing with division and modular reduction. However, proposals like those made by Brickell [3], Barret [1], and Montgomery [11] help to solve somehow these problems and they are widely cited by many authors in the literature.

After this, it can be said that the Montgomery modular multiplication is the most efficient algorithm for modular multiplication, so the proposal made in this paper will deal with it.

The regular procedure used in this algorithm begins with a translation of the conventional representation of positive integers and brings back this translation to its original conventional integer representation at the end of the multiplication procedure.

Besides, Montgomery modular multiplication replaces the trial division with a series of additions and divisions by a power of 2, this makes it suitable to be implemented in programmable devices, like Field Programmable Gate Array (FPGA) [20, 2, 9, 19, 14]. Divisions by the power of two can be made by making only shifts to the right.

This considerably reduces the consumption of resources on the programmable device. In the search for solving issues like those mentioned above, many authors have proposed some

Algorithm 1 Montgomery multiplication

```
Require: $m = (m_{n-1}, \cdots, m_0)_b$,
    $x = (x_{n-1}, \cdots, x_0)_b$, $y = (y_{n-1}, \cdots, y_0)_b$, with
    $0 \leq x, y < m$, $R = b^n$ with $\gcd(m, b) = 1$, and
    $m' = -m^{-1} \mod b$
Ensure: $A = xyR^{-1} \mod m$
1: $A \leftarrow 0$ {with $A = (a_n, a_{n-1}, \cdots, a_1, a_0)$}
2: for $i$ from $0$ to $n-1$ do
3:     $u_i \leftarrow (a_0 + x_i y_0)m' \mod b$
4:     $A \leftarrow (A + x_i y + u_i m)/b$
5: end for
6: if $A \geq m$ then
7:     $A \leftarrow A - m$
8: end if
9: return $A$
```

Algorithm 2 Montgomery multiplication with no final subtraction

```
Require: $m = (m_{n-1}, \cdots, m_0)_b$,
    $x = (x_{n-1}, \cdots, x_0)_b$, $y = (y_{n-1}, \cdots, y_0)_b$, with
    $0 \leq x, y < 2m$, $2m < R = b^n$ with $\gcd(m, b) =$
    $1$, and $m' = -m^{-1} \mod b$
Ensure: $A = xyR^{-1} \mod m$
1: $A \leftarrow 0$ {with $A = (a_{n-1}, \cdots, a_1, a_0)$}
2: for $i$ from $0$ to $n-1$ do
3:     $u_i \leftarrow (a_0 + x_i y_0)m' \mod b$
4:     $A \leftarrow (A + x_i y + u_i m)/b$
5: end for
6: return $A$
```

architectures, for example, Karatsuba based Montgomery modular multiplication [4, 7], Carry Save Adders (CSA) [8, 18, 6], Compact Signed Digits (SD) [15], and Systolic Architectures [2, 9, 19, 14, 17, 21, 5], to speed up the modular multiplication.

Specifically, the proposal Karatsuba based Montgomery modular multiplication is a high speed modular multiplication, it requires a few clock cycles compared with other proposals. However, it requires a large consumption of dedicated multipliers and resources. CSA is an interesting proposal since it uses only digital logic and no dedicated multipliers.

However, it requires a large consumption of resources and clock cycles to do the modular multiplication. Among these proposals, the Systolic Architecture is particularly interesting because it has a balance between the two earlier proposals. It reduces the resources compared with CSA and reduces the dedicated multipliers compared with Karatsuba-based Montgomery modular multiplication. To do the mentioned before, Systolic Architecture uses regular blocks called Processing Elements (PEs).

For example, authors like Guilherme Perin et al. [14] , compared a high radix systolic architecture with a high radix multiplexed multiplication in an FPGA. Amine Mrabet et al. [12] proposed the implementation of the Coarsely Integrated Operand Scanning (CIOS) method of Montgomery modular multiplication using a two-dimensional array of PEs. Hence, an improvement of PEs is presented in this work, from which the associated process can be speeded up and the resources used in the FPGA are reduced as well. For this, a Montgomery modular multiplication is implemented that can deal with long integers within the finite field $GF(p)$, where p is a prime number digitally expressed between 512 and 2048 bits.

This will be explained in the next section. The rest of the paper is organized as follows: Section 2 reviews the Montgomery modular multiplication algorithm and shows the improvements made. Section 3 shows the architecture proposed for implementation in an FPGA. In section 4, it is implemented the proposed architecture in modular exponentiation for use in the RSA algorithm. In section 5, it is presented the results obtained of the implementation and it is presented a comparison with other works reported. Finally, in section 6, we make the conclusions of this work.

2 Montgomery Multiplication

To perform a modular multiplication, it must have a residue of a division of the multiplication of the two positive integers. However, the arithmetic division is an operation that consumes high resources and time in hardware and software implementation, to avoid this there is a proposal called Montgomery modular multiplication algorithm [11].

The Montgomery modular multiplication is an algorithm that can be used to perform the modular multiplication $x \cdot y \mod m$ without the need to divide by the modulus m. In the

Algorithm 3 Montgomery multiplication for FPGA

```
Require: $m = (m_{n-1}, \cdots, m_0)_b$,
    $x = (x_{n-1}, \cdots, x_0)_b$, $y = (y_{n-1}, \cdots, y_0)_b$, with
    $0 \leq x, y < 2m$, $2m < R = b^n$ with $\gcd(m, b) =$
    $1$, and $m' = -m^{-1} \mod b$
Ensure: $A = xyR^{-1} \mod m$
1: $A \leftarrow 0$ {with $A = (a_{n-1}, \cdots, a_1, a_0)$}
2: for $i$ from $0$ to $n-1$ do
3:     Take the lowest $k$ bits of $u_i \leftarrow (a_0 + x_i y_0)m'$
4:     for $j$ from $0$ to $(n-1)$ do
5:         $(c_j, a_j) \leftarrow (a_j + x_i y + u_i m + c_j) \gg k$
6:     end for
7: end for
8: return $A$
```

Algorithm 4 Montgomery multiplier controller

```
1: Wait until the multiplication is enabled. $i = 0$.
2: Enable the operations of PEs.
3: $i = i + 1$.
4: if $i \geq n - 1$ then
5:     Go to state 9.
6: else
7:     Go to state 2.
8: end if
9: return the result.
```

next subsections, we will explain the Montgomery modular multiplication from the software and hardware point of view.

2.1 Software-base Montgomery Multiplication

First, from the software point of view, Montgomery modular multiplication implementations use Algorithm 1, shown next, which is the original algorithm [10]. This algorithm is the basis of many RSA software and hardware implementation systems.

In Algorithm 1, the operands x and y are positive integers with a radix b. The result is placed on A and after n iterations, it is equal to $xyR^{-1} \mod m$, which must retrieve the result $xy \mod m$.

Initially, it is required that x and y must be in the Montgomery domain, this is done applying the same Algorithm 1, with $\tilde{x}$ and R^2 as operands for x value and $\tilde{y}$ and R^2 as operands for y value, where $\tilde{x}$ and $\tilde{y}$ are the original operands for the modular multiplication. Next, to get the correct result $\tilde{A}$, it is needed to apply an additional Montgomery Multiplication with A and 1 as operands.

In modular exponentiation these added operations are inexpensive since they are done one time after the whole exponentiation. Since the mathematician Peter L. Montgomery published his algorithm in 1985, a lot of improvements were proposed by much research. The next section explains one of these approaches for hardware implementation.

2.2 Hardware-based Montgomery Modular Multiplication

On the other side, from the hardware point of view, FPGAs have been widely used to perform readily modular multiplications. It can be found proposals using several techniques that can improve Montgomery Multiplication implemented in FPGA [11, 20, 13]. One of these proposals is particularly interesting because it avoids the final subtraction used in the original algorithm [20]. To achieve this, x and y are set to be less than $2m$, and $2m$ must be less than b^n. This proposal is shown in Algorithm 2.

As was mentioned earlier, there is a technique that was proposed for the implementation that performs the Montgomery modular multiplication. This technique, called Systolic Architecture, uses a regular array based on a basic elemental PE. Therefore, an advantage is taken from the fact that operands are represented in a radix b of power of 2, this is $b = 2^k$, where k is the number of bits. This simplifies the operations in the digital implementation into the FPGA.

Each PE internally contains multipliers and adders, which manage large operands in a multi-precision context, based on Algorithm 2. Specifically, PEs are settled in a one dimensional array, all being identical, this was first proposed by Tenca et al. [17].

Their Montgomery multiplier has a scalable architecture, and it is based on the Multiple Word Radix-2 Multiplication Algorithm (MWR2MM). As was mentioned before, a Systolic Architecture can reduce the resources needed for the hardware implementation. Besides, it is possible to increase

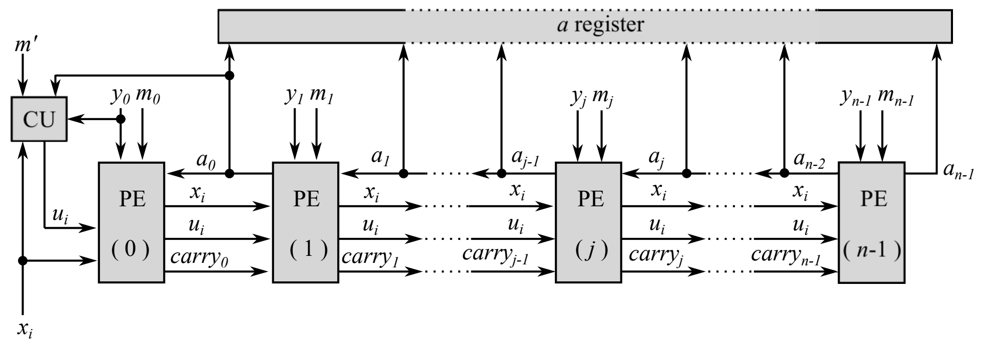

Fig. 1. Block diagram of systolic architecture

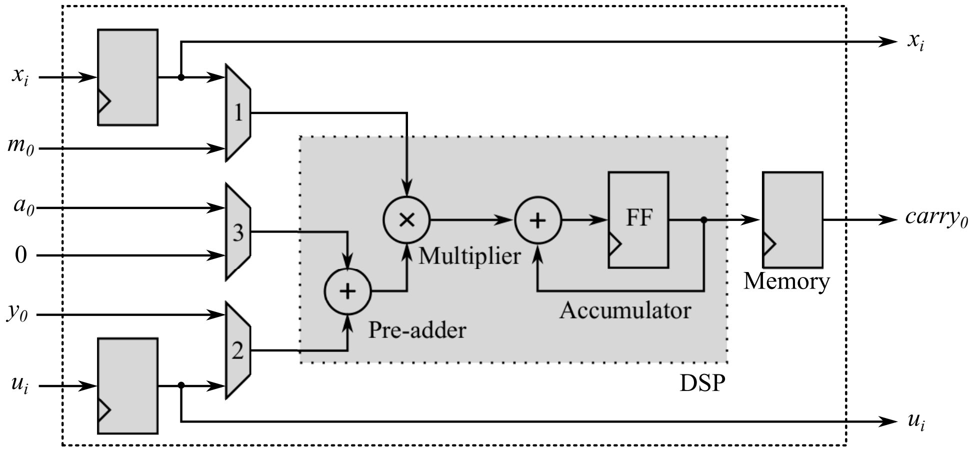

Fig. 2. Block diagram of initial processing element (IPE)

the number of bits of the operands adding only the required PEs for the specific dimension.

3 Systolic Architecture Proposed

Talking about the structure of the implemented system, a systolic architecture consists on a one-dimensional array of PEs [9, 20, 13], most of them are identical, the only different PEs are the first one and the last one, this will be explained Ylater.

The proposed architecture is based on Algorithm 2, where the operands are divided into n words having a length of k-bits. This is shown in Algorithm 3. Based on what was mentioned above, this shows that the total number of PEs in the systolic architecture has the same number as the total of words considered, this is, there will be n PEs.

The array of PEs performs step 4 outlined in Algorithm 2, this is, $A \leftarrow (A + x_i y + u_i m)/b$, where A has been represented with the same radix as the rest of the operands. This step is performed by steps 4 and 5 in Algorithm 3.

Since the systolic architecture works in a multi-precision context, each PE is responsible to perform the arithmetic operations of each word involved in the equation, step 5 in Algorithm 3.

The value c_j in step 3 stands for the carry obtained during the operation of each PE. As it was already said, the operands are represented in a radix b, this is, the numerical basis is the power of 2. So, the division by b of step 4 of Algorithm 2

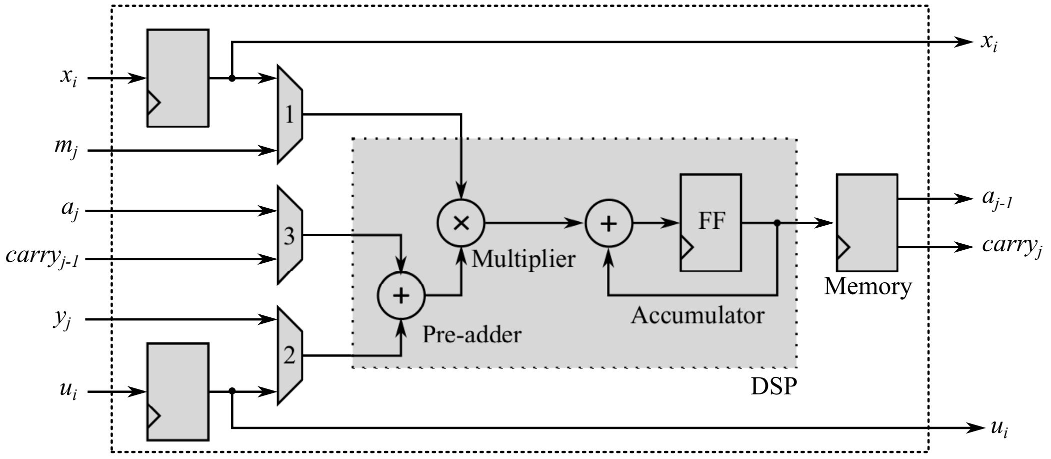

Fig. 3. Block diagram of general processing element (GPE)

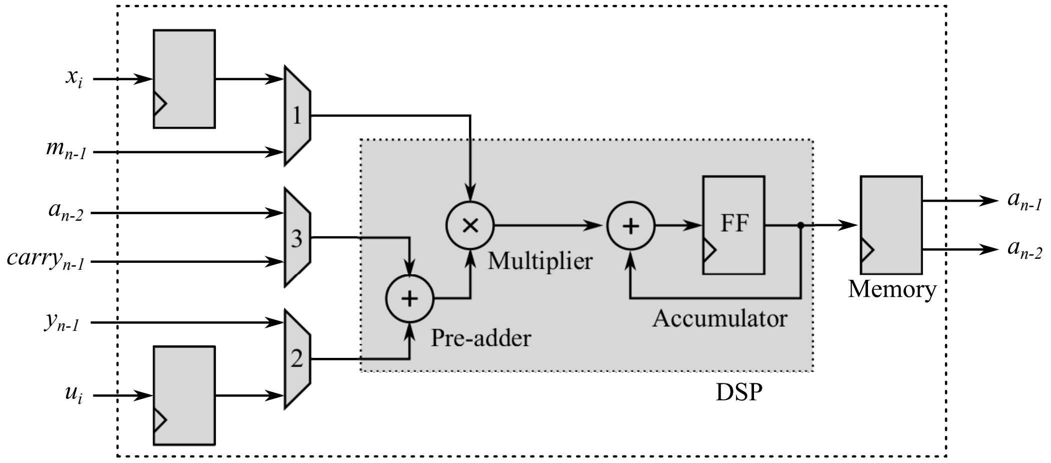

Fig. 4. Block diagram of final processing element (FPE)

is performed by a right shift operation of one word of k bits in step 5 of Algorithm 3.

The operation of step 3, this is, $u_i \leftarrow (a_0 + x_i y_0)\ m'$, in Algorithm 3, is performed in a separated block, called CU. To perform the inner loop of the Algorithm 3, a Finite State Machine (FSM) was implemented, called Montgomery Multiplier Controller (MMC).

The MMC block is designed to supply the corresponding words to the one-dimensional array of PEs and the block CU. The logic implementation of the MMC is shown in Algorithm 4 4. The MMC implemented works as follows:

– State 1: Variable i is initialized to 0 and waits until the multiplication is enabled. At the same time, y and m are split into n words of radix b and are sent to each of the corresponding PEs.

– State 2: The first PE is enabled while at the same time this PE enables the second PE, and this sequence is followed until the last PE is enabled.

– State 3: Here, i increase its account by 1, with the help of the counter. In this state, the next word in the register x is retrieved.

– State 4: In this state, i is compared with $n - 1$ and if it is greater or equal then goes to state 9,

Algorithm 5 Square and multiply always, left to right

```
Require: $m = (m_{l-1}, \cdots, m_0)_b$, $R = b^l$, $m' = -m^{-1} \mod b$, $e = (e_t, \cdots, e_0)_2$, with $e_t = 1$, and an integer $x < m$.
Ensure: $x^e \mod m$
 1: $S[0] \leftarrow R \mod m$
 2: $S[1] \leftarrow 0$
 3: $\tilde{x} \leftarrow \mathrm{MontMult}(x, R^2 \mod m, m)$
 4: for $i$ from $t$ down to 0 do
 5:     $S[0] \leftarrow \mathrm{MontMult}(S[0], S[0], m)$
 6:     $S[1] \leftarrow \mathrm{MontMult}(\tilde{x}, S[0], m)$
 7:     $S[0] \leftarrow S[e_i]$
 8: end for
 9: $S[0] \leftarrow \mathrm{MontMult}(S[0], 1, m)$
10: return $S[0]$
```

otherwise goes to state 2.

– State 9: This state ends the cycle and returns the result of the multiplication.

While states are performing within the MMC, Block CU is calculating at the same time the value of u_i. Fig 1 shows the block diagram of systolic architecture as was implemented in an FPGA.

In Fig 1., the value j shows the corresponding PE. In this work, it has been proposed three different types of PEs, which are based on step 5 of Algorithm 3. These PEs are Initial Processing Element (IPE), General Processing Element (GPE), and Final Processing Element (FPE). These PEs have been settled in a one-dimensional array as originally proposed.

To reduce resources, and area, and speed up the implementation of the PEs, it has been used Digital Signal Processing slices (DSP) integrated into the FPGA. For our implementation in the family Artix 7 of Xilinx, it is used DPS48E1. The main features of these DSPs are that they have internally a 25 $\times$ 18 two-complement multiplier and a 48-bit accumulator.

This DSP indicates that the maximum radix that can be implemented is with 18 bits, however, in this work 16 bits were used. Description of the elements in Fig. 1 is as follows: PE(0) is an IPE, PE($n-1$) is equivalent to an FPE and finally, the rest of PEs correspond to the type of GPE; then, there will be $n-2$ GPEs. Now, each type of PEs is explained in the next subsections:

3.1 Initial Processing Element

This first type of PE is called Initial Processing Element (IPE) and has direct communication with the MMC block. It receives an enabling signal, together with the value of x_i and u_i. The values x_i and u_i are saved in a register. Fig. 2 shows the block diagram implemented.

The performance of the IPE block is controlled by an FSM, which consists of three states. The performance is explained as follows: During the first state, x_i and u_i are saved in registers, and at the same time they are sent to the first GPE. Upon the reception of the enabling signal, the fist GPE is enabled and proceeds to the next state.

While yet in this first state, values of m_0 and u_i are sent to the multiplier across the multiplexers 1 and 2 respectively, also a value of 0 is sent to the pre-adder integrated into the DSP across the multiplexer 3 and then the result is saved in a register into a block called Accumulator of the DSP.

Next, during the second state the multiplication of x_i and y_0 is performed and it is added the value a_0 and the result is added to the earlier value of state 1 using the Accumulator of the DSP. Finally, when the third sate is reached, the results of the multiplications with the adders are stored into the block Memory. The result of this addition has a length of $2k+2$ bits.

However, the first k bits are discarded, this means the division by b in $A \leftarrow (A + x_i y + u_i m)/b$, and the rest of the bits are saved in a register and sent to the next PE as a carry. After this it is reset the accumulator of the DSP. Since operation time is important from the performance point of view, it should be mentioned that passing through each one of the states takes one clock cycle.

To save resources and speed up the operations, a DSP has been used as a 16×16 bits multiplier, pre-adder, and an accumulator.

ISSN 2007-9737

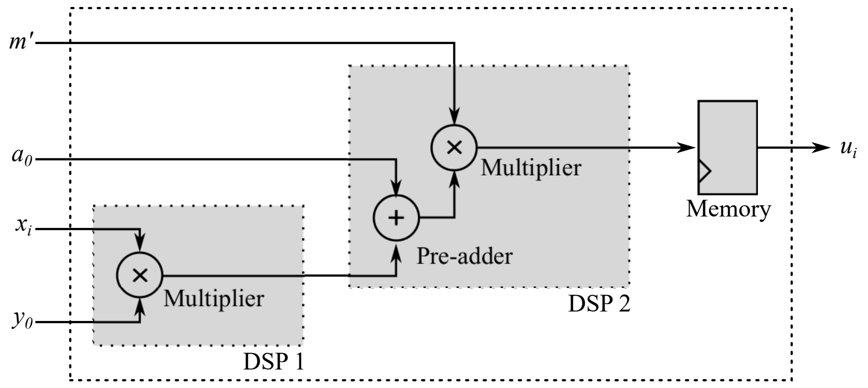

Fig. 5. Block diagram of calculus of u_i (CU)

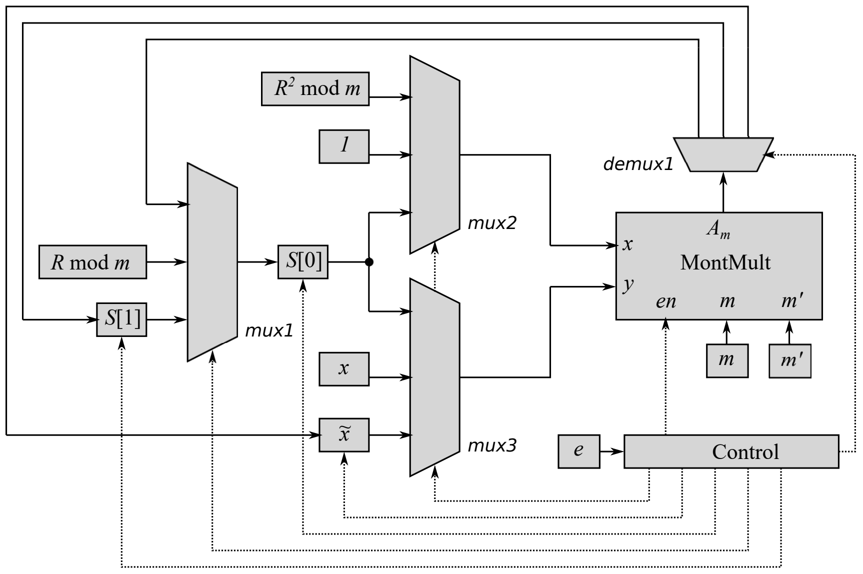

Fig. 6. Block diagram of the SaMAL2R exponentiation using montgomery multiplication

3.2 General Processing Element

The second type of PE is called General Processing Element (GPE). The GPE unlike the IPE receives as input a carry and gives the output a_{j-1}. Fig. 3 shows the block diagram of a GPE implemented into the FPGA. The GPE is replicated $n-2$ times, the first GPE receives the input carry from the IPE, and this first GPE sends its carry to the second GPE, this is made until the last GPE is reached and is then when the respective carry is sent to the last type of PE. The performance of this GPE is like the IPE, which is controlled by an FSM with three states. However, in the first state it is performed an addition of the carry input value.

The result, the same as in an IPE, has a length of $2k+2$ bits. The first k bits are equivalent to a_{j-1} and the rest of the bits are the output carry. The output a_{j-1} of each GPE is left-shifted to complement the operation of the division by b, as was mentioned in the description of the IPE.

3.3 Final Processing Element

The third and last type of PE is the Final Processing Element (FPE). The performance of this type of

Table 1. Resources used by the FPGA for the montgomery modular multiplication with a module of 512 bits

Device Utilization			
Resources	**Utilization**	**Available**	**Utilization %**
Slice LUTs	3098	63400	4.89 %
Slice Registers	2670	126800	2.11 %
Slice	1471	15850	9.28 %
DSPs	34	240	14.17 %

Table 2. Resources used by the FPGA for the montgomery modular multiplication with a module of 1024 bits

Device Utilization			
Resources	**Utilization**	**Available**	**Utilization %**
Slice LUTs	6736	63400	10.62 %
Slice Registers	5337	126800	4.21 %
Slice	3429	15850	21.63 %
DSPs	66	240	27.50 %

element is like GPE, however, the output carry of this PE is avoided. The block diagram of this PE is shown in Fig. 4. As we mentioned before, the output carry is avoided, and the output of this FPE has a length of $2k$ bits. The first k bits of the output are called a_{j-2} and the last k bits are called a_{j-1} which is equivalent to the value a_{n-1}. As the IPE and GPE, this kind of PE is controlled by an FSM with three states.

3.4 Block CU

Finally, the block named CU calculates the value u_i of step 3 as pointed out in Algorithm 2. As specified previously, the numerical basis is the power of 2, so for this block, the $\mod b$ operation needs only the LSB of k in the result. Fig. 5 shows the block diagram implemented for Block CU, where two DSPs were used to perform the multiplications.

4 Modular Exponentiation

In this section, modular exponentiation, which uses the modular multiplier proposed, is implemented into an FPGA. The modular exponentiation implemented is the Square and Multiply Always, Left to Right (SMAL2R). Algorithm 5 shows the regular modular exponentiation Square and Multiply Always algorithm and it is in a left-to-right form. This form begins the exponentiation with the most significant bit (MSB) of the exponent e and ends with the least significant bit (LSB).

From Algorithm 5, it is required that $\gcd(m, R) = 1$. The values of $R \mod m$ and $R^2 \mod m$ may be provided as inputs. As was mentioned at the beginning, the operands for the Montgomery modular multiplication were settled into the Montgomery domain, this is done in step 1 and at the end the result was retrieved in step 9 of the Algorithm 5. The function called $\mathrm{MontMult}()$ in Algorithm 5 performs the Montgomery modular multiplication. In this function, the first and the second parameter are the operands and finally the third parameter is the modulus.

An FSM was implemented to control the Montgomery modular exponentiation. In Fig. 6, it is shown the general block diagram which shows the implementation in FPGA. In Fig. 6 the block

Table 3. Resources used by the FPGA for the montgomery modular multiplication with a module of 2048 bits

Device Utilization			
Resources	**Utilization**	**Available**	**Utilization %**
Slice LUTs	12854	63400	20.27 %
Slice Registers	10648	126800	8.40 %
Slice	5132	15850	32.38 %
DSPs	130	240	54.17 %

Table 4. Comparison of hardware resources and performance for the montgomery modular multiplication

Work	Device	Module m (bits)	Radix b (bits)	Freq. (MHz)	Clock Cycles	Slice LUTs	Slice Registers	Slices	DSPs
Perin	Virtex-4	1024	16	110	384	-	-	7012	130
Perin	Virtex-5	1024	16	130	384	-	-	6642	130
Wang	Virtex-5	1024	16	120	199	14440	7826	-	66
C.McIvor	Virtex-2	1024	16	104	199	-	-	5709	131
Mrabet	Artix-7	1024	16	65	66	5242	4208	2072	161
Mrabet	Virtex-5	1024	16	65	66	5824	6072	-	-
Proposed architecture	Artix-7	1024	16	100	194	6736	5337	3429	66

labeled as Control is the FSM which controls the whole modular exponentiation according to the exponent e. The blocks labeled as $S[0]$, $S[1]$ and $\tilde{x}$ are registers working as memories, and they are set and reset according to the block labeled as Control.

The block labeled as MontMult in Fig. 6 holds the modular multiplication proposed. The operands for this block are set by the block Control. RSA algorithm was chosen to provide both the public key and the private key. From the RSA algorithm, the value e, also known as public-key in Montgomery modular exponentiation, is used to encrypt, and x is the data to be encrypted, also known as plain text. This is shown in 1:

$$\mathrm{enc} = x^e \mod n. \quad (1)$$

To decrypt the data encrypted, the same modular exponentiation is used, however, the exponent is now d, also known as private-key, in this case, the value x is known cipher-text. This is shown in 2:

$$x = \mathrm{enc}^d \mod m. \quad (2)$$

Since e or d are represented as a binary with $t+1$ bits, there will be $t+1$ cycles as it was mentioned before. Now, considering that Square-and-multiply Always, Left to Right Algorithm is regular, no matter if the corresponding bit of the exponent during the cycle is one or zero, the two multiplications will be performed in the loop as well. Thus, the total number of multiplications during the complete process of the exponentiation is:

$$\mathrm{multiplications} = 2(t+1) + 2. \quad (3)$$

The two extras multiplications in 3 are due to steps 3 and 9 in Algorithm 5. Once the blocks were defined and the function of each one was explained, implementation on a FPGA was made and results are presented in the following section.

5 Results

The proposed design was implemented into an FPGA Artix-7 XC7A100T-CSG324 working at 100 MHz; no area or speed optimization was set for

Table 5. Resources used by the FPGA for the montgomery exponentiation with a module of 1024 bits

Device Utilization			
Resources	**Utilization**	**Available**	**Utilization %**
Slice LUTs	7587	63400	11.97 %
Slice Registers	9499	126800	7.49 %
Slice	4253	15850	26.83 %
DSPs	66	240	27.50 %

the synthesis. The synthesis into the FPGA for the proposed design was settled to a module m of 512, 1024, and 2048 bits of length, and the radix b of 16 bits and performance of the multiplications were evaluated. Besides, the final subtraction was avoided settling the conditions $x < 2m$ and $y < 2m$.

First, Table 1 shows the resources used for the proposed implementation of a module of 512 bits. In this case, the implementation used 1 IPE, 1 FPE, and 30 GPE's. Table 2 shows the resources used for the proposed implementation using a module of 1024 bits. The implementation used 1 IPE, 1 FPE, and 62 GPEs. Finally, Table 3 shows the resources used for the proposed implementation using a module of 2048 bits. The implementation used 1 PE, 1 FPE, and 126 GPEs.

Additionally, the implementation takes 98 clock cycles to perform the Montgomery modular multiplication for a module of 512 bits, 194 clock cycles for a module of 1024 bits, and 386 clock cycles for a module of 2048 bits. Table 4 shows a comparison of this work with other implementations using 1024 bits modular multiplication, and a radix of sixteen bits.

From the above, it can be seen that the proposal allows a reduction of resources consumption within the programmable device, compared with other developments. On the other side, the implementation of the Montgomery modular exponentiation uses a module m of a length of 1024 bits.

The resources used by the implementation into the FPGA for the modular exponentiation are shown in Table 5. As it was already said, in this proposal, the implementation of the Montgomery modular exponentiation was used to encrypt and to decrypt data.

The public-key and the private-key were generated with the RSA algorithm. According to Table 4, the proposed architecture reduces 5 clock cycles compared with the performance reported in[9] and [21]. With these 5 clocks cycles, the time consumption in the Modular Exponentiation of Algorithm 5 is reduced.

For example, with $t = 1023$ bits and using 3, there will be a total of 2,050 multiplications. So, this means that for the works [9] and [21] this process takes a total of 407,950 clock cycles while the proposed architecture here reported takes a total of 397,700 clock cycles, having considerable differences of 10,250 clock cycles. Therefore, time reduction with this algorithm is evident. Therefore, time reduction with this algorithm is clear. This proposal compared to that Mrabet [12] reduces the number of DSPs used, hower, it requires a greater number of clock cycles.

6 Conclusions

As can be seen in the tables reported, the most resources in the modular exponentiation are due to the modular multiplication, so it is important to reduce them. As we can see from the results of this work, the proposed architecture, compared with the other works that use a high-radix modular multiplication reduces the time consumption and the resources used for the FPGA, as well.

Due to this low use of resources into the FPGA, the implementation of the Modular Exponentiation and Multiplication using the proposed Algorithm can allow its use in programmable devices that have limited available resources. Therefore, the programmable device to be used will be cheaper than a high-end programmable device.

This work implements modular exponentiation for RSA cryptographic algorithms; however, it is possible to use in other cryptographic algorithms like ECC, ElGamal, DH. Even more it is possible to implement this proposal to different modular exponentiation algorithms. Besides, it is important to say also that the proposed architecture is scalable to another module. For example, making, this same procedure with a device that has more DSPs, it will be possible to increase the module to a value up-to 4,096 bits with the use of 258 DSPs.

Acknowledgments

Funding. This work was supported by CINVESTAV - IPN and the projects PAPIIT IA102323 and CI2410 of the UNAM and FESC.

Data availability. Data sharing does not apply to this article as no datasets were generated or analyzed during the current study.

References

1. **Barrett, P. (1987).** Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. Advances in Cryptology, Springer Berlin Heidelberg, pp. 311–323. DOI: 10.1007/3-540-47721-7_24.
2. **Blum, T., Paar, C. (1999).** Montgomery modular exponentiation on reconfigurable hardware. Proceedings of the 14th IEEE Symposium on Computer Arithmetic, pp. 70–77. DOI: 10.1109/ARITH.1999.762831.
3. **Brickell, E. F. (1983).** A fast modular multiplication algorithm with application to two key cryptography. Advances in Cryptology, Springer US, pp. 51–60. DOI: 10.1007/978-1-4757-0602-4_5.
4. **Chow, G. C. T., Eguro, K., Luk, W., Leong, P. (2010).** A karatsuba-based montgomery multiplier. Proceedings of the International Conference on Field Programmable Logic and Applications, pp. 434–437. DOI: 10.1109/FPL.2010.89.
5. **Gowda, L., Shaila, K., Venugopal, K. R. (2016).** Elliptic curve cryptography implementation on FPGA using montgomery multiplication for equal key and data size over GF(2m) for wireless sensor networks. Proceedings of the IEEE Region 10 Conference, pp. 468–471. DOI: 10.1109/TENCON.2016.7848043.
6. **Kuang, S. R., Wu, K. Y., Lu, R. Y. (2016).** Low-cost high-performance VLSI architecture for montgomery modular multiplication. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 24, No. 2, pp. 434–443. DOI: 10.1109/TVLSI.2015.2409113.
7. **Liu, R., Li, S. (2019).** A design and implementation of montgomery modular multiplier. Proceedings of the IEEE International Symposium on Circuits and Systems, pp. 1–4. DOI: 10.1109/ISCAS.2019.8702684.
8. **McIvor, C., McLoone, M., McCanny, J. V. (2003).** Fast montgomery modular multiplication and RSA cryptographic processor architectures. Proceedings of the 37th Asilomar Conference on Signals, Systems Computers, Vol. 1, pp. 379–384. DOI: 10.1109/ACSSC.2003.1291939.
9. **McIvor, C., McLoone, M., McCanny, J. V. (2005).** High-radix systolic modular multiplication on reconfigurable hardware. Proceedings of the IEEE International Conference on Field-Programmable Technology, pp. 13–18. DOI: 10.1109/FPT.2005.1568518.
10. **Menezes, A. J., van-Oorschot, P. C., Vanstone, S. A. (2018).** Handbook of applied cryptography. CRC press.
11. **Montgomery, P. L. (1985).** Modular multiplication without trial division. Mathematics of computation, Vol. 44, No. 170, pp. 519–521.
12. **Mrabet, A., El-Mrabet, N., Lashermes, R., Rigaud, J. B., Bouallegue, B., Mesnager, S., Machhout, M. (2017).** A scalable and systolic architectures of montgomery modular

multiplication for public key cryptosystems based on DSPs. Journal of Hardware and Systems Security, Vol. 10076, pp. 219–236. DOI: 10.1007/978-3-319-49445-6_8.

13. **Orup, H. (1995).** Simplifying quotient determination in high-radix modular multiplication. Proceedings of the 12th Symposium on Computer Arithmetic, pp. 193–199. DOI: 10.1109/ARITH.1995 .465359.

14. **Perin, G., Gomes, D., Martins, J. B. (2011).** Montgomery modular multiplication on reconfigurable hardware: Systolic versus multiplexed implementation. International Journal of Reconfigurable Computing, Vol. 2011, pp. 1–10. DOI: 10.1155/2011/127147.

15. **Rezai, A., Keshavarzi, P. (2017).** Compact SD: A new encoding algorithm and its application in multiplication. International Journal of Computer Mathematics, Vol. 94, No. 3, pp. 554–569. DOI: 10.1080/00207160.2015.1119269.

16. **Rivest, R. L., Shamir, A., Adleman, L. (1978).** A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, Vol. 21, No. 2, pp. 120–126. DOI: 10.1145/359340.359342.

17. **Tenca, A. F., Koc, C. K. (2003).** A scalable architecture for modular multiplication based on montgomery's algorithm. IEEE Transactions on Computers, Vol. 52, No. 9, pp. 1215–1221. DOI: 10.1109/TC.2003.1228 516.

18. **Verma, R., Dutta, M., Vig, R. (2016).** FPGA implementation of RSA based on carry save montgomery modular multiplication. Proceedings of the International Conference on Computational Techniques in Information and Communication Technologies, pp. 107–112. DOI: 10.1109/ICCTICT.2016.7 514561.

19. **Walter, C. D. (1993).** Systolic modular multiplication. IEEE Transactions on Computers, Vol. 42, No. 3, pp. 376–378. DOI: 10.1109/12.210181.

20. **Walter, C. D. (1999).** Montgomery exponentiation needs no final subtractions. Electronics Letters, Vol. 35, No. 21, pp. 1831–1832.

21. **Wang, P., Liu, Z., Wang, L., Gao, N. (2013).** High radix montgomery modular multiplier on modern FPGA. Proceedings of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, pp. 1484–1489. DOI: 10.110 9/TrustCom.2013.180.

Article received on 27/06/2022; accepted on 16/07/2024.
**Corresponding author is José de Jesús Morales Romero.*