Novel algorithms and hardware architectures for Montgomery Multiplication over GF(p)

M. Morales-Sandoval[a], A. Diaz Perez[a]

[a]*Laboratorio de Tecnologias de la Informacion. CINVESTAV-Tamaulipas. Parque Tecnotam, Victoria, Tamps, 87130, Mexico*

Abstract

This report describes the design and implementation results in FPGAs of a scalable hardware architecture for computing modular multiplication in prime fields GF(p), based on the Montgomery multiplication (MM) algorithm. Starting from an existing digit-serial version of the MM algorithm, a novel *digit-digit* based MM algorithm is derived and two hardware architectures that compute that algorithm are described. In the proposed approach, the input operands (multiplicand, multiplier and modulus) are represented using as radix $\beta = 2^k$. Operands of arbitrary size can be multiplied with modular reduction using almost the same hardware since the multiplier's kernel module that performs the modular multiplication depends only on k. The novel hardware architectures proposed in this paper were verified by modeling them using VHDL and implementing them in the Xilinx FPGAs Spartan and Virtex5. Design trade-offs are analyzed considering different

Email addresses: `mmorales@tamps.cinvestav.mx` (M. Morales-Sandoval), `adiaz@tamps.cinvestav.mx` (A. Diaz Perez)

This is a technical report. Part of the information presented here is appearing in the journal article *"Scalable GF(p) Montgomery Multiplier based on a digit-digit computation approach"*, to be published in the journal *IET Computers and Digital Techniques.* in 2015

operand sizes commonly used in cryptography and different values for k. The proposed designs for MM are well suited to be implemented in modern FPGAs, making use of available dedicated multiplier and memory blocks reducing drastically the FPGA's standard logic while keeping an acceptable performance compared with other implementation approaches. From the Virtex5 implementation, the proposed MM multiplier reaches a throughput of 242Mbps using only 219 FPGA slices and achieving a 1024-bit modular multiplication in 4.21μsecs.

Keywords: Montgomery algorithm, Digit-based Multiplier, FPGA, Finite Field

1. Introduction

Arithmetic in the finite field GF(p) is crucial in modern cryptography for real-life applications such as security protocols in computer networks or software applications for secure data storage. These high level applications for data security rely on cryptographic algorithms such as RSA [1], a public key cryptosystem that performs encryption and digital signature processes based on exponentiation on GF(p). Another example is Elliptic Curve Cryptography for elliptic curves defined over GF(p) [2, 3]. Several GF(p) multiplications are required to achieve a single GF(p) exponentiation, being this operation the most time consuming and the RSA's bottleneck. This fact has motivated the creation of algorithms and custom hardware architectures to compute these arithmetic operations faster in order to improve the overall performance of high level data security mechanisms. One of the most efficient GF(p) multiplication algorithms is the Montgomery one.

The Montgomery multiplication algorithm [4] takes as input two integers X, Y of size n-bits and the modulus p such that $2^{n-1} < p < 2^n$. The result is $Z = X \times Y \times 2^{-n} \bmod p$ with $\gcd(2^n, p = 1)$. This last condition is easily satisfied since p is actually a prime number. The *p-residue* of a number A is defined as $A' = A \times 2^n \bmod p$. The Montgomery multiplication of two *p-residue* numbers A' and B' is another *p-residue* number C' that corresponds to the integer $C = A \times B \bmod p$. Thus, it is necessary to convert the numbers from the ordinary domain (with ordinary multiplication) to the Montgomery domain and backwards. Given the number A in the ordinary domain, its corresponding number A' in the Montgomery domain is computed as the Montgomery multiplication $A' = A \times 2^{2n} \bmod p$. On the other hand, given C' in the Montgomery domain, its corresponding number in the ordinary domain is also computed as the Montgomery multiplication $C = C' \times 1 \bmod p$. Despite the conversion cost, a notorious advantage is achieved if many Montgomery multiplications are required, as it is the case of encryption in RSA.

The Montgomery multiplier has the advantage of being able to perform modular multiplications where the modulus is of a general form and also at very low area [5]. In the literature there are several versions of the Montgomery algorithm and hence different hardware proposals for computing a Montgomery multiplication. Three approaches can be distinguished from the reported works: full parallel, bit-serial, and digit-based multipliers.

Full parallel multipliers. Having the n-bit operands X, Y, and p, Z is computed in a single clock cycle. This way the resulting multiplier would exhibit a very high throughput but also a long critical path and high area resources. A representative work implementing this kind of multiplier is [6]. Recent works as [7] use the Karatsuba approach to decompose the multiplication recursively until the size of the partial product multiplications can be computed by the embedded multiplier blocks in modern FPGAs.

Bit-serial multipliers. The algorithm corresponding to this kind of multiplier is known as the Radix-2 Montgomery Multiplication algorithm, requiring n iterations to compute Z. The operand X is parsed iteratively bit per bit while Y and p are accessed in parallel (all their bits are accessed at a time). This kind of multiplier is the simplest for hardware implementation. However, the involved operations in the MM algorithm are performed with full precision of Y and p having an intrinsic limitation: once a hardware design based on that algorithm is defined for n bits, it cannot work with other precision [8]. Moreover, being Y and p large numbers (i.e. greater than 1024 bits) the hardware multipliers results with large wire delays and complex routing, which also affects the maximum delay and thus the multiplier's performance. Recent works have provided implementation results of this kind of multiplier, for example [5].

Digit-based multipliers. The digit-based approach has been better preferred and studied since it allows to compromise area/performance according to specific implementation requirements. Further, a digit-oriented algorithm allows to develop scalable hardware units for the Montgomery multiplication algorithm, that is, hardware architectures able to work for any operands precision. In the literature, three types of digit-based Montgomery multiplier are distinguished. The first one, named in this paper as the High-Radix (2^k) Montgomery algorithm ($R2^kMM$) parses the operand X

considering w-bits at a time while all bits of operands Y and p are accessed at a time. A representative work implementing this algorithm is [9]. Compared to a bit-serial multiplier, performance is gained since the number or iterations in the algorithm is reduced from n to $\lceil n/w \rceil$. However, considering more than one bit at a time from X implies more area resources in the design. The second digit-based approach for Montgomery multiplication is to decompose the operands Y and p in several digits of size w and parse them iteratively while X is parsed bit-by-bit. The corresponding algorithm for this approach was named in [8] as the Multiple-word Radix-2 Montgomery Algorithm (*MWR2MM*). Of course, this second approach will lead to a design that sacrifices performance since more iterations are required to parse each digit from Y and p but it provides more regular hardware architectures that lead to multipliers with higher clock frequencies. The most studied hardware architectures for the *MWR2MM* algorithm have been the systolic ones [10, 11, 12, 13]. The third approach is the Multiple-word High-Radix (2^k) Montgomery Multiplication algorithm ($MWR2^kMM$) presented in [14]. Under this algorithm, k determines the number of bits from X processed at a time while the operands Y and p are parsed in words of size w. For this kind of multiplier, designs only for $k = 2$ [13] and $k = 3$ [14] have been proposed due to the enormous cost in area.

This work presents a novel digit-based Montgomery multiplier. Different to the previous approaches, this contribution presents as main distinctive characteristic a *digit-digit approach* for constructing novel Montgomery multipliers, splitting the three operands X, Y and p in digits of size k and computing a Montgomery multiplication like in software but exploiting the parallelism in instructions. Although the *digit-digit* Montgomery algorithm proposed in this work uses the same principle than the $MWR2^kMM$ algorithm reported in [14, 10], it differs in the following: *i*) the *digit-digit* algorithm is derived from the Walter's digit-serial algorithm [15] that does not require final subtraction, it is not derived from the Radix-2 Montgomery multiplication algorithm . *ii*) the Booth encoding is not used for each digit from X as it is done in [14] neither a shift register is used for storing X as it occurs in [10]. *iii*) While in the $MWR2^kMM$ algorithm X is split in digits of size k and Y, p are split in digits of size w, in our proposed algorithm the three operands are split in digits of size k. *iv*) The corresponding hardware architecture for the proposed *digit-digit* algorithm is not a systolic array of processing elements as it is the case of [14, 10] or other similar works as [11, 12, 13].

This report is an extension of the previous work published in [16]. Compared to that earlier work, this paper provides an extended explanation of the novel digit-digit algorithm for Montgomery multiplication called IDDMM, providing comparisons with other Montgomery algorithms in the literature. Also, this extended version presents two hardware architectures for computing the IDDMM algorithm, discussing implementation results on FPGAs. A parametric design for the multipliers described in VHDL allowed to determine the best configuration for the digit size k and obtain the best performed architectures that compromise area and performance (efficiency). FPGAs are the ideal choice as computing platform to carry out this study. Under the proposed *digit-digit approach* it is posible to have from very compact designs (sacrificing performance) to better performer multipliers (at expenses of more area resources).

The next section introduces the Montgomery algorithm and a review of different versions of it in the literature. Section 3 describes the novel *digit-digit* Montgomery multiplication algorithm proposed in this work, comparing it against existing algorithms. Section 4 describes in detail two hardware architectures implementing the *digit-digit* Montgomery multiplier, whose complexity depends only on k, not on the operands size. Section 5 presents, discusses and compares the achieved results against related works. Finally, Section 6 concludes this work and points out future work.

2. Montgomery multiplication algorithm

The Montgomery algorithm originally reported in [4] performs the operation $X \times Y \times R^{-1} \bmod p$, given $X, Y < p$ and R such that the greatest common denominator $(p, R) = 1$. The integer R usually is a power of 2 so the requirement $gcd(p, R) = 1$ is satisfied if p is odd. Let modulus p be an n-bit number with $2^{n-1} \leq p < 2^n$ and $R = 2^n$. The Radix-2 Montgomery multiplication algorithm (R2MM) depicted in figure 1 is the simplest method for computing $Z = X \times Y \times 2^{-n} \bmod p$. $X = (x_{n-1}, \cdots, x_1, x_0)$ is parsed from the least to the most significant bit. At each iteration, the partial result stored in S is updated according to the parity of $S + x_iY$. If $S + x_iY$ is even, the division at line 5 in algorithm 1 is easily computed as a logical right shift operation. Contrary, if that value is odd, the addition of p to $S + x_iY$ does not affect the sum (which is reduced modulo p) but makes the result $S + x_iY + p$ be an even number that can be again easily divided by 2. Clearly, the latency of algorithm R2MM is n. At line 10, a subtraction could

```
procedure R2MM($X, Y, p$)
    $S \leftarrow 0$
    for $i \leftarrow 0$ to $n - 1$ do
        if $(S + x_i Y)$ is even then
            $S \leftarrow (S + x_i Y)/2$
        else
            $S \leftarrow (S + x_i Y + p)/2$
        end if
    end for
    if $S \geq p$ then
        $S \leftarrow S - p$
    end if
end procedure
```

Figure 1: Radix-2 Montgomery multiplication algorithm (**R2MM**).

be required if the final result S is greater or equal to p.

In 1999, Tenca and Koc [8] presented the word-based Radix-2 algorithm for Montgomery multiplication named MWR2MM and listed in figure 2. For this version, X is parsed bit-per-bit as in the R2MM algorithm but the operands Y, p are split in $e = \lceil n/w \rceil$ words each of size w and parsed iteratively. Words from Y and p are denoted as $Y = [0, Y^{(e-1)}, \cdots, Y^{(1)}, Y^{(0)}]$, and $p = [0, p^{(e-1)}, \cdots, p^{(1)}, p^{(0)}]$ respectively. Observe that Y, p are extended by one most significant word equal to zero. As a convention with other related works, words from operands are denoted by superscripts while individual bits are denoted by subscripts.

In the R2MM algorithm, the parity of $S + x_i Y$ is tested using only the first bit of Y and S. However, in the MWR2MM algorithm at line 4, the parity of $S + x_i Y$ is tested considering only the first word of Y and S. $x_i Y^{(0)} + S^{(0)}$ is stored in $S^{(0)}$ and the resulting carry is stored in C_a. The parity of $S + x_i Y$ will be given by testing the bit $S_0^{(0)}$. In the MWR2MM algorithm the operation

```
**procedure** MWR2MM($X, Y, p$)
    $S \leftarrow 0$
    **for** $i \leftarrow 0$ to $n-1$ **do**
        $(C_a, S^{(0)}) \leftarrow x_i Y^{(0)} + S^{(0)}$
        **if** $S_0^{(0)} = 1$ **then**
            $(C_b, S^0) \leftarrow S^{(0)} + p^{(0)}$
            **for** $j \leftarrow 1$ to $e$ **do**
                $(C_a, S^{(j)}) \leftarrow C_a + x_i Y^{(j)} + S^{(j)}$
                $(C_b, S^{(j)}) \leftarrow C_b + p^{(j)} + S^{(j)}$
                $(S^{(j-1)}) \leftarrow (S_0^{(j)}, S_{w-1\cdots 1}^{(j-1)}$
            **end for**
        **else**
            **for** $j \leftarrow 1$ to $e$ **do**
                $(C_a, S^{(j)}) \leftarrow C_a + x_i Y^{(j)} + S^{(j)}$
                $(S^{(j-1)} \leftarrow (S_0^{(j)}, S_{w-1\cdots 1}^{(j-1)})$
            **end for**
        **end if**
        $S^{(e)} \leftarrow 0$
    **end for**
**end procedure**
```

Figure 2: Multiple Word Radix-2 Montgomery multiplication algorithm (**MWR2MM**).

$\left(S + x_iY + S_0^{(0)}p\right)/2$ is computed iteratively parsing the digits form S, Y and p. The total amount of iterations is $e + 1$.

For the case when $S_0^{(0)} = 1$ one digit of the result is computed at each iteration j in two steps. First, the operation $C_a + x_iY^{(j)} + S^{(j)}$ is performed, where C_a is the carry of the same operation in the previous iteration $j-1$. In the second step, the operation $C_b + p^{(j)} + S^{(j)}$ is performed, also considering the carry C_b obtained from the same operation in the previous iteration $j - 1$. At this point, a new digit $S^{(j)}$ of size w of the result S is ready. The division by 2 is accomplished by a shift to the right operation on the previous computed digit $S^{(j-1)}$ using as fill value the least significant bit from the current digit $S^{(j)}$. In case $S_0^{(0)} = 0$ the operation $(S+x_iY)/2$ is computed iteratively considering only the digits from S and Y, computing a new digit of the result S. This is shown in line 14 of algorithm MWR2MM, where C_a stores the carry obtained from the sum $C_a + x_iY^{(j)} + S^{(j)}$ and used in the computation of $S^{(j+1)}$ in the next iteration. The latency of the MWR2MM is $n \times (\lceil n/w \rceil + 1)$.

In 2001, the Multiple-Word High-Radix (Radix-2^k) Montgomery Multiplication MWR2^kMM algorithm was proposed [14]. In this new version shown in figure 3, not only the operands Y and p are split in several digits of size w but also the operand X. Since Radix-2^k is used, X is parsed in groups of k bits at a time. To author's knowledge, only hardware architectures for this algorithm using $k = 2$ (Radix-4) [17, 13] and $k = 3$ (Radix-8) [14] have been reported in the literature. The MWR2^kMM algorithm is a variant of the word-based FIOS algorithm for Montgomery multiplication for software implementation [18]. The idea in the MWR2MM algorithm was to compute

S at each iteration i having as a result $S_0^{(0)} = 0$ in order to perform the division by 2 as a shift to the right operation. In the MWR2^kMM algorithm the requirement is to have at each iteration i the k least significant bits of S equal to zero, that is, $S_{k-1,\ldots,0}^{(0)} = 0$ in order to perform a division of S by 2^k as a right shift operation of k-bits. Under this approach the number of iterations i is reduced to $\lceil n/k \rceil$ and it is necessary to compute q^i at each iteration i such that $S+X^{(i)}\times Y+q^i p$ is divisible by 2^k. Authors in [13] only considered $k = 2$ (Radix-4) in their reported hardware systolic implementation and remarked that the case $k > 2$ is impractical. For that reason, a customized function implemented in a look-up-table is designed. However, theoretically any k value can be used having as only requirement that $S^{(0)} + X^{(i)}Y^{(0)} + q^i p^{(0)} = 0$ (mod 2^k). In [14], q^i is computed as $[[S^{(0)} + X^{(0)}Y^{(0)}] \times (-p_{k-1,\ldots0}^{(0)^{-1}})] \bmod 2^k$.

The Montgomery algorithm based on a *digit-digit* computation proposed and implemented in this work is described in the next section. Although it follows the same principle of the MWR2^kMM algorithm its derivation and implementation is different.

3. Novel *digit-digit* Montgomery multiplier

In 1999, C. Walter proposed an improved iterative algorithm (IMM) listed in figure 4 for computing a Montgomery multiplication [15, 19]. Compared with the original Montgomery algorithm, algorithm in figure 4 performs one extra iteration, making the conditional final subtraction unnecessary. The improved algorithm is therefore time-constant and avoids the implementation of a subtracter [20]. The notation used from here on in the computation of $X \times Y \times R^{-1} \bmod p$, with $R = \beta^{n+1}$, is shown in table 1.

```
1:  procedure MWR$2^k$MM$(X, Y, p)$
2:      $S \leftarrow 0$
3:      $x_{-1} \leftarrow 0$
4:      for $j \leftarrow 0$ to $n-1$ step $k$ do
5:          $q_Y \leftarrow Booth(x_{j+k \cdots j-1})$
6:          $(C_a, S^{(0)}) \leftarrow S^{(0)} + (q_Y * Y)^{(0)}$
7:          $q_p \leftarrow S^{(0)}_{k-1 \cdots 0} * (2^k - p^{(0)^{-1}}_{k-1 \cdots 0}) \bmod 2^k$
8:          $(C_b, S^{(0)}) \leftarrow S^{(0)} + (q_M M)^{(0)}$
9:          for $i \leftarrow 1$ to $\lceil n/w \rceil$-1 do
10:             $(C_a, S^{(i)}) \leftarrow C_a + S^{(j)} + (q_Y * Y)^{(i)}$
11:             $(C_b, S^{(i)}) \leftarrow C_b + S^{(j)} + (q_p * p)^{(i)}$
12:             $(S^{(i-1)}) \leftarrow (S^{(i)}_{k-1 \cdots 0}, S^{(i-1)}_{w-1 \cdots k}$
13:         end for
14:         $C_a \leftarrow C_a$ or $C_b$
15:         $S^{(\lceil n/w \rceil)} \leftarrow signExt\left(C_a, S^{(\lceil n/w \rceil - 1)}_{w-1 \cdots k}\right)$
16:     end for
17:     if $S \geq p$ then
18:         $S \leftarrow S - p$
19:     end if
20: end procedure
```

Figure 3: Multiple Word Radix-2^k Montgomery multiplication algorithm (**MWR2^kMM**).

```
Require: integers $X = \sum_{i=0}^{n} X_i \beta^i$, $Y = \sum_{i=0}^{n} Y_i \beta^i$ and $p = \sum_{i=0}^{n-1} p_i \beta^i$, with
    $0 < X, Y < 2p$, $R = \beta^n$ with $\gcd(p, \beta) = 1$, and $p' = -p^{-1} \bmod \beta$
Ensure: $A = \sum_{i=0}^{n-1} a_i \beta^i = X \times Y \times R^{-1} \bmod p$
1:  procedure IMM$(X, Y, p)$
2:      $A^{<0>} \leftarrow 0$
3:      for $i \leftarrow 0$ to $n$ do
4:          $q^{<i>} \leftarrow (A^{<i>} + X_0 \times Y_i) \times p' \bmod \beta$
5:          $A^{<i+1>} \leftarrow ([A^{<i>} + X \times Y_i] + q^{<i>} \times p)/\beta$
6:      end for
         return $A^{<i+1>}$
7:  end procedure
```

Figure 4: Iterative Montgomery multiplication algorithm (**IMM**) without final subtraction.

N	Operands size (bits)
n	Number of k-bit digits of modulus and operands
p	The modulus defining the prime field GF(p)
X	Multiplier operand in the modular multiplication
Y	Multiplicand operand in the modular multiplication
$A^{<i>}$	Result of $X \times Y \times R^{-1} \bmod p$ at the end of iteration i
p_j	k-bit digit from p during the iteration j
X_j	k-bit digit from X during the iteration j
Y_i	k-bit digit from Y during the iteration i
$A_j^{<i>}$	k-bit digit from $A^{<i>}$ during the iteration j
β	Radix ($\beta = 2^k$)
$q^{<i>}$	Constant during iteration i for computing $A^{<i+1>} = A^{<i>} + X \times Y_i + q^{<i>} \times p$
p'	Precomputed value, being $p' = -p^{-1} \bmod \beta$
$C^{<j>}$	k-bit carry at iteration j

Table 1: Notation

In algorithm IMM it is assumed that the numbers X, Y, p with the restriction $0 \leq X, Y < 2p$ are expressed as n symbols using $\beta = 2^k$ as radix, and represented as in equations (1-3). In algorithm IMM, the most significant digit $Y_n = 0$. After $(n + 1)$ iterations, algorithm IMM computes $A = X \times Y \times R^{-1} \bmod p = \sum_{j=0}^{n-1} A_j \beta^j$. The number p' satisfies $pp' + RR^{-1} = 1 \bmod p$ [4]. Actually, p' is a k-bit constant value during all the multiplication algorithm so it that can be computed in advance and fixed for a given value k and finite field GF(p).

$$p = (p_{n-1} \cdots p_0)_\beta = \sum_{i=0}^{n-1} \beta^i p_i \tag{1}$$

$$X = (X_{n-1} \cdots X_0)_\beta = \sum_{i=0}^{n-1} \beta^i X_i \tag{2}$$

$$Y = (Y_{n-1} \cdots Y_0)_\beta = \sum_{i=0}^{n-1} \beta^i Y_i \tag{3}$$

At step 5 of algorithm IMM, division by β is implemented as a right shift operation since the k-least significant bits of $A^{<i>} + X \times Y_i + q^{<i>} \times p$ are equal to zero. At each iteration i, both $q^{<i>}$ and $A^{<i+1>}$ are computed. From a sequential approach, these two operations could be performed by the set of operations described in equations 4 - 10.

$$M = X \times Y_i \tag{4}$$

$$T = A_0 + X_0 \times Y_i \tag{5}$$

$$q^{<i>} = T_0 \times p' \bmod \beta \tag{6}$$

$$R = p \times q^{<i>} = (R_n R_{n-1} \cdots R_1 R_0) \tag{7}$$

$$S = A^{<i>} + M = (S_n S_{n-1} \cdots S_1 S_0) \tag{8}$$

$$U = R + S = (U_n U_{n-1} \cdots U_1 U_0) \tag{9}$$

$$A^{<i+1>} = U/\beta = (U_n \cdots U_1) \tag{10}$$

Since $q^{<i>}$ is the result of a modulo β reduction, just the k least significant bits (LSB) from T (T_0) are considered for the computation of $q^{<i>}$.

The multiplications leading to the $(n+1)$-digit numbers M, $q^{<i>}$ and R are implemented for example, using the schoolbook method (shift and add), processing k-bits at a time from X (when computing M) and p (when computing R). Using the polynomial representation of X in radix β, the operation $M = X \times Y_i$ can be expressed as in equation 11. Each term $(X_j \times Y_i)$ is of size $2k$-bits. The accumulative sum of terms in equation 11 is shown graphically in figure 5.

$$\begin{aligned} X \times Y_i &= \left(\sum_{j=0}^{n-1} \beta^j X_j\right) \times Y_i \\ &= \sum_{j=0}^{n-1} \beta^j (X_j \times Y_i) \\ &= \beta^n (X_{n-1} \times Y_i) + \cdots + \beta (X_1 \times Y_i) + (X_0 \times Y_i) \end{aligned} \tag{11}$$

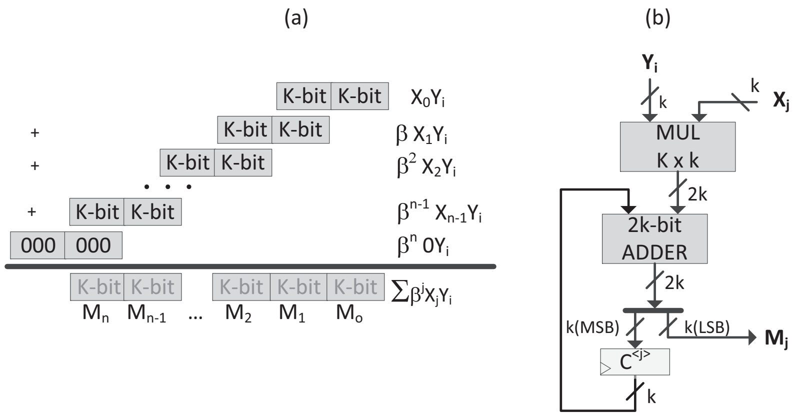

Figure 5: Iterative computation of $X \times Y_i$ parsing each digit X_j from X. (a) Partial products and final sum for computing $X \times Y_i$. (b) Block diagram of the circuit for computing $X \times Y_i$ using the schoolbook method.

Thus, a new digit M_j of the result is obtained as each digit X_j is processed. A $2k$-bit adder is required to perform the accumulative sum of every partial product. Also, a k-bit register $C^{<j>}$ is required to store the k-bit carry for the sum in the next iteration. At the beginning, this register is set to zero. It can be demonstrated that the result from the adder in figure 5 b) fits in $2k$-bits since the maximum value form the adder is $(2^k-1)*(2^k-1)+2^k-1 = 2^{2k}-2^k$ which is less than $2^{2k}-1$. At each iteration j, the $2k$-bit result from the adder is divided in two parts. The least significant k bits M_j are taken as a digit of the result M and the k most significant bits are used as the carry value $C^{<j>}$ for the next iteration. Observe in figure 5 a) that the last k-bit digit being part of the multiplication is stored in the $C^{<j>}$ register after processing the n-th digit from X. By inserting an additional most significant digit to X ($X_n = 0$) the last digit M_n of the multiplication M is correctly obtained. An alternative solution is to use a multiplexer that selects either M_j or $C^{<j>}$ as the resulting digit of the multiplication. So, during the first n iterations the selected value from the multiplexer would be M_j and only during the last iteration $(n+1)$ (with $X_n = 0$) the selected value by the multiplexer would be $C^{<j>}$. Independently of the solution selected, the latency of the multiplier in figure 5 b) is $(n+1)$ for computing $X \times Y_i$. Under the same analysis, the hardware module in figure 5 b) can be used to compute $R = q^{<i>} \times p$ also in $(n+1)$ iterations, processing a digit p_j from p at a time.

Despite the fact that in a multiplication the smaller operand is considered as the multiplier for reducing the number of terms in the sum, in this work it is considered the contrary; X and p are considered as the multiplier to compute M and R iteratively. This allows to use a smaller multiplier, that in this case is a $(k \times k)$-multiplier. This comes with a penalization in the latency.

Consider now the computation of variable S in equation 8. Instead of waiting that all the terms of M be computed to calculate S, both computations can be performed in parallel. According to equations 12-14, since $S = A^{<i>} + X \times Y_i$, each digit of A and X can be processed at the same time, replacing the 2-input adder in figure 5 b) by a 3-input adder and adding each digit $A_j^{<i>}$ to $X_j \times Y_i + C^{<j>}$. The result of the 3-input adder again is at most a $2k$-bit value, since the maximum result from the 3-input adder is $2^{2k} - 2^k + 2^k - 1 = 2^{2k} - 1$.

$$S = A^{<i>} + X \times Y_i = \sum_{j=0}^{n} \beta^j A_j^{<i>} + \left(\sum_{j=0}^{n} \beta^j X_j \right) \times Y_i \tag{12}$$

$$= \sum_{j=0}^{n} \beta^j A_j^{<i>} + \sum_{j=0}^{n} \beta^j (X_j \times Y_i) \tag{13}$$

$$= \sum_{j=0}^{n} \beta^j (A_j^{<i>} + (X_j \times Y_i)) \tag{14}$$

Again, S will be computed one digit at a time resulting in the sequence of digits $(S_n S_{n-1} \cdots S_1 S_0)$. According to this, the first digit S_0 is actually T_0 in equation 6. Thus, $q^{<i>} = S_0 \times p' \bmod \beta$. Being p' a fixed number, the circuit for this multiplication could be simplified.

During the same clock cycle for computing S_0, $q^{<i>}$ can be also computed since the involved logic is combinational. After that clock cycle, the computation of variable R in equation 7 can be launched using a multiplier as the one shown in figure 5 b), parsing the digits p_j from p one at a time. This multiplier would produce the sequence $(R_n R_{n-1} \cdots R_1 R_0)$ requiring $(n+1)$ clock cycles for computing $q^{<i>} \times p$. Of curse, this multiplier will have its own carry register, say $C_R^{<j>}$.

Having the digits S_j and R_j at each clock cycle, the computation of U in equation 9 can be also performed one digit at a time. In this case, the operation $(S_j + R_j)$ is a $(k+1)$-bit number. The k least significant bits conform one digit U_j of U while the remaining most significant bit of $(S_j + R_j)$ is stored in a register $C_U^{<j>}$ and used as 1-bit carry for the sum $(S_{j+1} + R_{j+1})$ in the next iteration. Since $U = A^{<i>} + X \times Y_i + q^{<i>} \times p$ will result in a number having the k least significant bits equal to zero, the first digit U_0 could be simply discarded in order to have equation 10 computed as $A^{<i+1>} = U/\beta = (U_n, \cdots, U_1)$.

The digits U_j obtained as described previously will be the digits $A_j^{<i+1>}$ of $A^{<i+1>}$ used in the next iteration $(i+1)$ (see algorithm 2 and equation 10). At the beginning of each cycle j the digit $A_j^{<i>}$ is read and used to compute S_j. At the end of iteration j the value $A_j^{<i>}$ is no longer used and it must be overwritten by U_{j+1}. This overwriting is denoted as $A_j^{<i+1>} = U_{j+1}$. In the iteration $j = n-1$, the last digit $A_{n-1}^{<i>}$ of $A^{<i>}$ is read and used to compute U_{n-1} using the digit X_{n-1}. In the next and final iteration $j = n$, $A_n^{<i>} = 0$ and $X_n = 0$. So in this last iteration $S_n = 0 + 0 \times Y_i + C_S^{<n-1>} = C_S^{<n-1>}$ without generating a new carry $(C_S^{<n>} = 0)$. Also during this last iteration $p_n = 0$ and $R_n = q^i \times 0 + C_R^{<n-1>} = C_R^{<n-1>}$, with $C_R^{<n>} = 0$. Hence, $U_n = C_S^{<n-1>} + C_R^{<n-1>} + C_U^{<n-1>}$ without producing a carry $(C_U^{<n>} = 0)$. Thus, the last digit U_n of $A^{<i+1>}$ comprises the addition of all the final carries.

Based on the previous explanation, a novel iterative *digit-digit* Montgomery algorithm is proposed, named IDDMM and listed in algorithm 1.

The same assumptions for the IMM algorithm are kept.

Algorithm 1 Novel iterative *digit-digit* Montgomery algorithm (IDDMM)

```
Require: integers $X = (0, X_{n-1}, ..., X_0)$, $Y = (Y_{n-1}, ..., Y_0)$ and $p = (0, p_{n-1}, ..., p_0)$, with $0 < X, Y < 2p$, $R = \beta^{n+1}$ with $\gcd(p, \beta) = 1$, and $p' = -p^{-1} \bmod \beta$
Ensure: $A = \sum_{i=0}^{n-1} A_i\beta^i = X \times Y \times R^{-1} \bmod p$
 1: procedure IDDMM($X, Y, p$)
 2:     $A^{<0>} \leftarrow 0$
 3:     for $i \leftarrow 0$ to $n-1$ do
 4:         $C_S^{<0>} \leftarrow 0$
 5:         $C_R^{<0>} \leftarrow 0$
 6:         $C_U^{<0>} \leftarrow 0$
 7:         for $j \leftarrow 0$ to $n$ do
 8:             $\{C_S^{<j+1>}, S_j\} \leftarrow A_j^{<i>} + X_j \times Y_i + C_S^{<j>}$
 9:             if $j = 0$ then
10:                 $q^{<i>} \leftarrow (S_j \times p') \bmod \beta$
11:             end if
12:             $\{C_R^{<j+1>}, R_j\} \leftarrow q^i \times p_j + C_R^{<j>}$
13:             $\{C_U^{<j+1>}, U_j\} \leftarrow S_j + R_j + C_U^{<j>}$
14:             if $j > 0$ then
15:                 $A_{j-1}^{<i+1>} \leftarrow U_j$
16:             end if
17:         end for
18:     end forreturn $A^{<n+1>}$
19: end procedure
```

Table 2 shows a comparative of the proposed IDDMM algorithm against the well-known Montgomery multiplication algorithms reviewed in previous section. In that comparison, 2^k is the radix used and N is the number of bits of operands. Our proposed IDDMM could be directly compared against the MWR2^kMM since both of them parse the operands X, Y, p in several digits. At first sight, it could be perceived that our proposal is the special case of MWR2^kMM using $w = k$, however this is not true. The IDDMM algorithm

Algorithm	#Iterations	Is scalar HW impl. possible?	Notes
R2MM	N	No	Greater area consumption since all bits of Y, p are accessed in parallel.
MWR2MM	$N \times (\lceil N/w \rceil + 1)$	Yes	Y and p are split in $e = \lceil N/w \rceil words$. X is parsed bit per bit.
MWR2^kMM	$\lceil N/k \rceil \times (\lceil N/w \rceil + 1)$	Yes	for each k-bits processed from X, all the words from Y, p are accessed. Practical implementations have only considered $k = 2$ and $k = 3$.
IDDMM (proposed)	$\lceil N/k \rceil \times (\lceil N/k \rceil + 1)$	Yes	p, X, Y are split in digits of size k. No recoding technique is used as it is done in MWR2^kMM. Practical implementations with $k > 3$ are possible.

Table 2: A comparison of Montgomery multiplication algorithms for hardware implementation.

proposed in this work is derived from the digit-serial Montgomery algorithm proposed by C. Walter in [15, 19], where no final subtraction is required. As a result, several differences against the MWR2^kMM algorithm can be listed. For example, the MWR2^kMM algorithm still requires to test if the final result is greater than p and performs the final subtraction if required. Additionally, the MWR2^kMM algorithm uses Booth encoding for each digit parsed from X. In the IDDMM algorithm not any encoding technique is required. Another difference is that the division by $\beta = 2^k$ in algorithm MWR2^kMM is performed over each partial product of size $(k + w)$ whereas in algorithm IDDMM that division is performed by simply discarding U_0.

In the next section two hardware architectures are described for executing algorithm IDDMM.

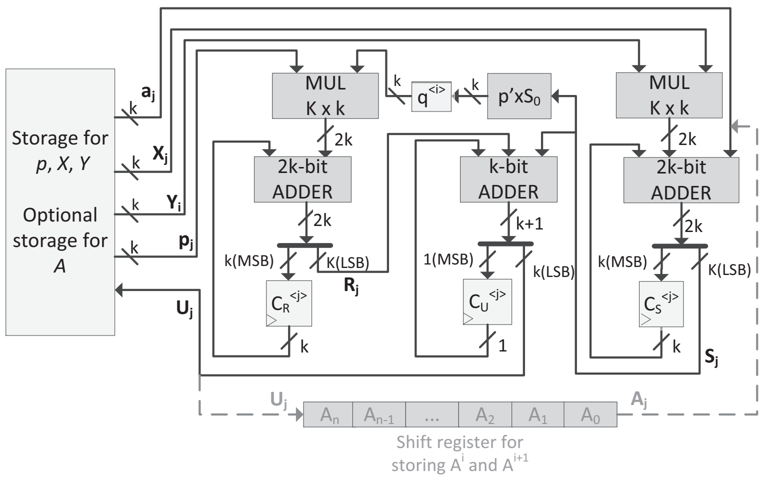

Figure 6: Architecture 1 for the IDDMM Montgomery algorithm.

4. Hardware architectures for the proposed IDDMM algorithm

Algorithm 1 described in previous section is directly mapped to hardware using as basic block the multiplier shown in figure 5 b). Since A, $C_R^{<j>}$, $C_U^{<j>}$ and $C_S^{<j>}$ are registers, the instructions 2, 4, 5 and 6 in algorithm 1 can be executed by asserting a reset signal. The instructions 8, 10, 12 and 13 involve combinatorial logic for computing the partial products and the accumulative additions. The first iteration $j = 0$ lasts two clock cycles. In the first one, the value $S_0 \times p'$ is loaded into the k-bit register $q^{<i>}$. In the second one, the correct values are taken for instructions 12 and 13, computing the first digit U_0. From there on, at every clock cycle a digit U_j is obtained. A hardware implementation of algorithm 1 is depicted in figure 6, which can be divided in a memory module for storing the operands p, X, Y and a datapath for performing all the required computations in algorithm 1.

The datapath in figure 6 has as basic blocks two $(k \times k)$-bit integer multipliers, two $2k$-bit adders, one k-bit adder, three k-bit registers and one 1-bit register. Being p' a constant value, the operation $p' \times S_0$ can be computed by a custom combinatorial circuit or another $(k \times k)$-bit integer multipliers could be used. There are two options to store and update variable A in algorithm 1. If A is stored in memory, it is required that at the beginning all the memory words be set to zero. The number of words to store A should be $n + 1$ with the $(n + 1)$-th word set to zero always. The first word $U_0 = 0$ is discarded (see explanation on this in previous section) and it is not stored in memory. Every next word $U_1, U_2, ..., U_n$ is stored in the words $A_0, A_1, ..., A_{n-1}$. This can be achieved by using a memory with synchronous writing and asynchronous reading. Another way to achieve the desired functionality for storing, accessing and updating the digits of A is to use a shift to the right register. Such a register will store n digits of size k-bits, shifting to the right k bits at each clock cycle and using the value U_j as the filling digit. Each digit A_j should be accessed through the k-least significant bits of A, that is, from A_0. However, even using the shift register to store A, at the end of the last iteration $i = n$, all the words in the shift register A representing the result of the Montgomery multiplication should be written back to memory. Independently of the way A is accessed and updated, the circuit in figure 6 has a latency of $n \times (n + 2)$. An additional clock cycle is required by architecture 1 when executing the inner loop of algorithm 1 in order to compute $q^{<i>}$ and store it in a register. After this, all the $n + 1$ digits form A, X and p will be parsed as it has been previously described.

An alternative hardware architecture for computing algorithm 1 is shown in figure 7. In this new architecture the $2k$-bit results from the $(k \times k)$ multipliers are used for computing U_j by means of a 3-input $2k$-bit adder

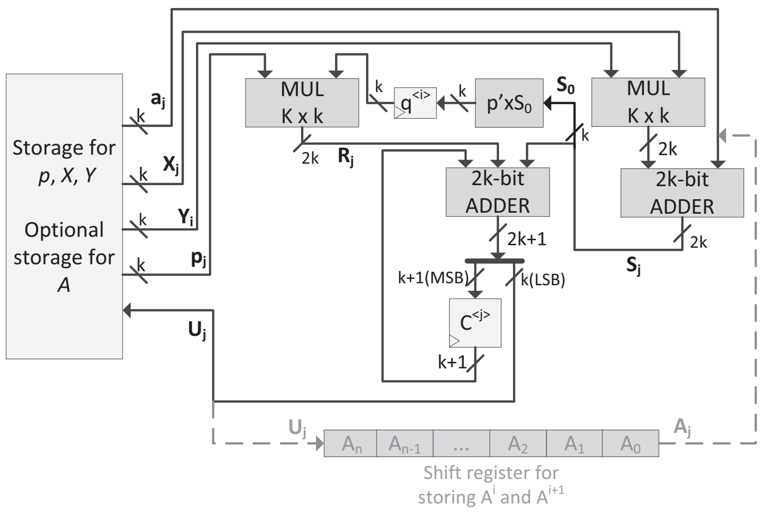

Figure 7: Architecture 2 for Algorithm 2, a variant of the IDDMM Montgomery algorithm.

instead of a k-bit adder. This way the carry values $C_S^{<j>}$, $C_R^{<j>}$ and $C_U^{<j>}$ are no longer required. Now, only one register is needed to store a single $(k+1)$-bit carry from the sum $(S_j + R_j + C^{<j>})$, being S_j and R_j of size $2k$ bits and $C^{<j>}$ of size $(k+1)$. The operation $(S_j + R_j + C^{<j>})$ fits in $(2k+1)$-bits. From this result, the k least significant bits conforms the result U_j and the remaining $(k+1)$ most significant bits are used as the carry $C^{<j>}$ for the next iteration. Again, during the last iteration $j = n$, the 3-input adder computes $0 + 0 + C^{<n-1>} = C^{<n-1>} = U_n$ which is a k-bit number. These changes are reflected in algorithm 2. In this new version of algorithm 1 the variables S and R store a $2k$-bit value, U_j is a k-bit number and the sum $S + R + C^{<j>}$ is written as $C^{<j+1>}\beta + U_j$. The latency of this second architecture is the same than the one for architecture 1.

The area complexity of the two previous hardware architectures for computing algorithms 1 and 2 is mainly determined by k in the radix $\beta = 2^k$

Algorithm 2 A variant of the proposed *IDDMM* Montgomery algorithm computed by architecture 2.

```
Require: integers $X = (0, X_{n-1}, ..., X_0)$, $Y = (Y_{n-1}, ..., Y_0)$ and $p = (0, p_{n-1}, ..., p_0)$, with $0 < X, Y < 2p$, $R = \beta^n$ with $\gcd(p, \beta) = 1$, and $p' = -p^{-1} \bmod \beta$
Ensure: $A = \sum_{i=0}^{n} A_i \beta^i = X \times Y \times R^{-1} \bmod p$
 1: procedure IDDMM($X, Y, p$)
 2:     $A^{<0>} \leftarrow 0$
 3:     for $i \leftarrow 0$ to $n$ do
 4:         $C^{<j>} \leftarrow 0$
 5:         for $j \leftarrow 0$ to $n$ do
 6:             $S \leftarrow A_j^{<i>} + X_j \times Y_i$
 7:             if $j = 0$ then
 8:                 $q^i \leftarrow (S_0 \times p') \bmod \beta$
 9:             end if
10:             $R \leftarrow q^i \times p_j$
11:             $\{C^{<j+1>}, U_j\} \leftarrow S + R + C^{<j>}$
12:             if $j > 0$ then
13:                 $A_{j-1}^{<i+1>} \leftarrow U_j$
14:             end if
15:         end for
16:     end for
         return $A^{<n+1>}$
17: end procedure
```

being used, not by the size of the operands. In the next section the hardware implementation of the main building blocks in figures 6 and 7 is described.

4.1. Adder and parallel multiplier blocks

The main computing blocks in architectures 1 and 2 are the $2k$-bit adder and the $(k \times k)$-bit multipliers. This section describes the hardware design of these blocks.

4.1.1. Parallel $(k \times k)$-bit multiplier

Consider the multiplication of two k-bit integer numbers u, v. A parallel multiplier should compute $u \times v$ in a single clock cycle multiplying each bit of u with w and adding the partial product of each multiplication to obtain the final result [21]. Representing the k-bit numbers u, v in radix-2 and supposing that k is a power of 2, u and v can be divided in two $(k/2)$-bit numbers u_1, u_2 and v_1, v_2 respectively such that equations 15–17 hold.

$$u = u_2 2^{k/2} + u_1 \tag{15}$$

$$v = v_2 2^{k/2} + v_1 \tag{16}$$

$$u \times v = (u_2 \times v_2)2^k + (u_1 \times v_2 + u_2 \times v_1)2^{k/2} + u_1 \times v_1 \tag{17}$$

Figure 8 shows graphically the parallel computation of $u \times v$ and the corresponding block diagram for computing equations 15–17. The divide and conquer strategy for multiplication was originally proposed by Karatsuba and not only GF(p) Mongomery multipliers have been proposed using these approach [22] but also multipliers for GF(2^m) [7, 23]. Each multiplier in

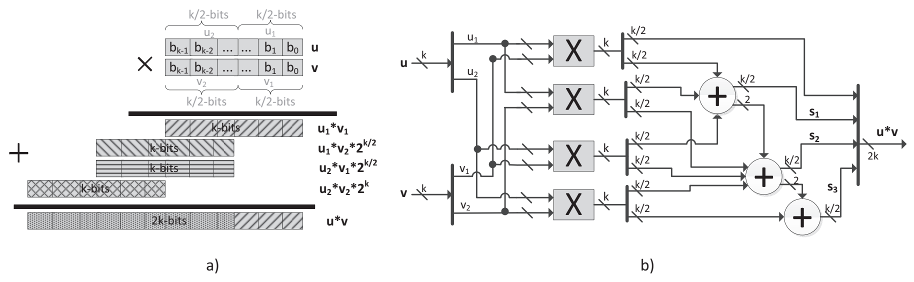

Figure 8: Parallel multiplier based on a divide and conquer approach.

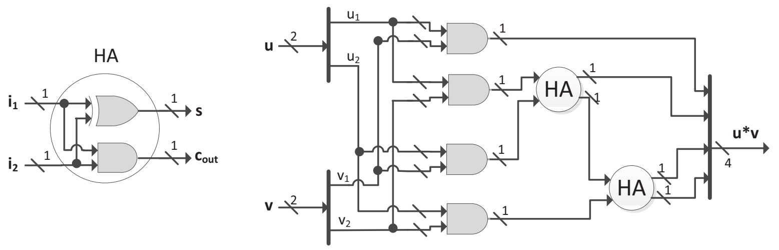

Figure 9: Basic parallel multiplier. a) Half adder. b) Parallel multiplier for $k = 2$ with simplified logic compared with figure 8 b).

figure 8 b) is defined in the same way. The recursively definition ends when the inputs to the parallel multiplier are of length 1 bit. The basic parallel multiplier that allows constructing bigger precision ones is designed for $k = 2$ and shown in figure figure 9. When $k = 2$ the multiplier blocks in figure 8 involve a the multiplication of 1-bit numbers. These multiplications produce a result of 1-bit, so the adders in figure 8 are simplified to half adders as it is shown in figure 9 b).

4.1.2. Adders

The parallel multiplier shown in figure 8 b) requires the three additions s_1, s_2, s_3. Instead of performing the addition of three numbers of $2k$ bits as it would be done according to figure 8 a), the addition is performed only with the needed bits using three adders of at most $(k/2)$-bits which can be implemented using a combination of full and half adders. The first addition s_1 involves three $(k/2)$-bit numbers resulting in a $k/2$-bit number and a 2-bit carry. The second addition s_2 involves three $(k/2)$-bit numbers plus the 2-bit carry from the computation of s_1 resulting again in a $k/2$-bit number and a 2-bit carry (the maximum resulting value from this addition is less than $2^{k/2+2}-1$). The third addition s_3 involves the sum of one $k/2$-bit number and the 2-bit carry from the computation of s_2 resulting in a $(k/2)$-bit number. The corresponding hardware architectures that compute these three sums are shown in figure 10.

The computation of s_2 requires more hardware resources whereas computing s_3 requires fewer. The circuit shown in figures 10c) and 10d) can be generalized to compute the additions of hardware architectures 1 and 2 in figures 6 and 7 respectively. For example, for computing the addition of a $2k$-bit number with a k-bit number, the circuit in figure 10 d) would require $k-1$ full adders instead of only one.

4.1.3. Computation of $p' \times S_0$

In figures 6 and 7 it is necessary to compute $p' \times S_0$, which is the multiplication of a k-bit variable number S_0 and a k-bit constant number p'. Since one of the operands is known in advance, some low level modules in the multiplier architecture shown in figure 8 can be removed or simplified using well known facts in boolean algebra (eg. $q = 1$ AND $q = 0$ OR q).

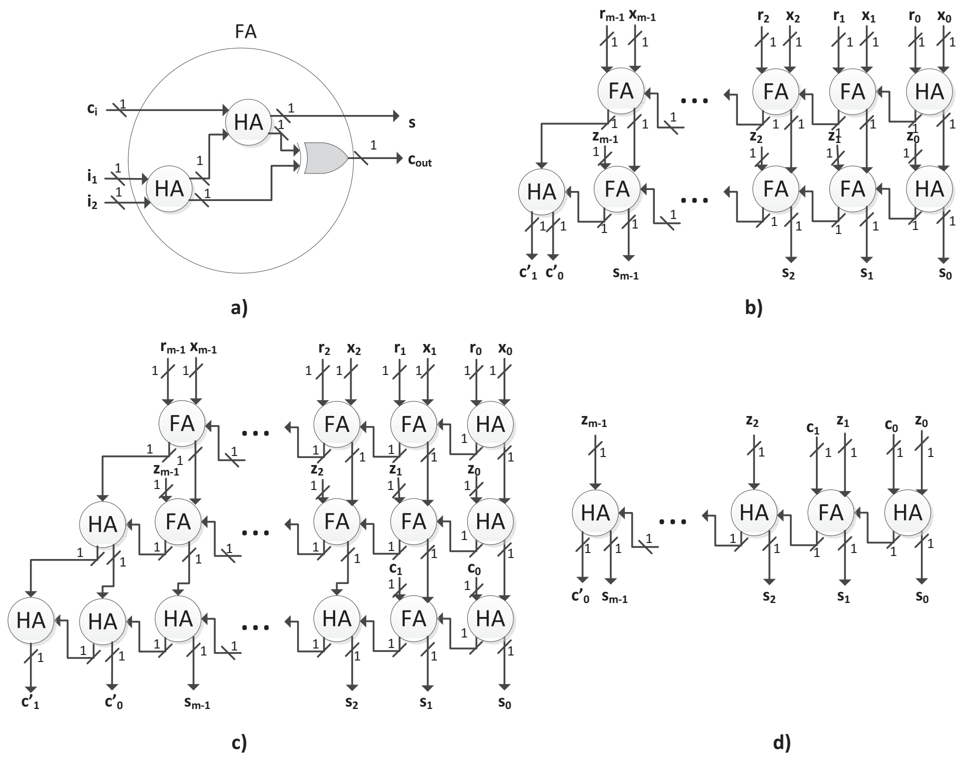
FA
HA
a)
b)
c)
d)
r_{m-1} x_{m-1}
r_2 x_2 r_1 x_1 r_0 x_0
z_{m-1} z_2 z_1 z_0
c_1 c_0
c'_1 c'_0 s_{m-1} s_2 s_1 s_0
c_i i_1 i_2 s c_{out}

Figure 10: Hardware implementation of adders required in figure 8 b) where $k/2 = m$. a) Basic full adder using the half adder (HA) in figure 9a). b), c) and d) are the circuits for computing s_1, s_2 and s_3 respectively in figure 8 b).

	Combinatorial resources	1-bit registers	Critical path	Clock cycles
Architecture 1	3MUL$_k$ + (4k-1)FA + (3k+5)HA.	$2k+1$	$D_{MUL} + 2D_{ADD_3}$	$n(n+2)$
Architecture 2	3MUL$_k$ + 4kFA + (2k+4)HA.	$k+1$	$D_{MUL} + D_{ADD_3} + D_{ADD_2}$	$n(n+2)$
Architecture 1 1 pipeline level	3MUL$_k$ + (4k-1)FA+ (3k+5)HA.	$7k+1$	$\max\{D_{MUL},2D_{ADD_3}\}$	$n(n+3)$
Architecture 2 1 pipeline level	3MUL$_k$ + 4kFA + (2k+4)HA.	$6k+1$	$\max\{D_{MUL},\ D_{ADD_3} + D_{ADD_2}\}$	$n(n+3)$
Architecture 1 2 pipeline levels	3MUL$_k$ + (4k-1)FA+ (3k+5)HA.	$11k+1$	$\max\{D_{MUL},D_{ADD_3}\}$	$n(n+4)$
Architecture 2 2 pipeline levels	3MUL$_k$ + 4kFA + (2k+4)HA.	$10k+1$	$\max\{D_{MUL},\ D_{ADD_3}\}$	$n(n+4)$

Table 3: Area and time complexity of proposed hardware architectures for the IDDMM algorithm.

4.2. Area and time complexity of IDDMM architectures

The hardware architectures 1 and 2 require a control unit to orchestrate the dataflow in algorithms 2 and 1. This control unit is implemented by a finite state machine (FSM) than resets, enables and disables the registers in the datapath as well as generates the addresses to memory blocks and their control signals (read, write, etc.). For simplicity, the logic associated to this FSM is not considered in the area complexity of the proposed Montgomery multipliers. This is usually done in related works for example [9]. In the same way, the memory resources to allocate the operands and the result is not considered.

Table 3 summarizes the complexity of architectures 1 and 2 in terms of area resources as well as the critical path. The area complexity is mainly determined by the $(k \times k)$-bit multipliers, each with area cost (MUL$_k$) given by:

$$\text{MUL}_k = 4{*}\text{MUL}_{k/2} + \text{S}_{1_{k/2}} + \text{S}_{2_{k/2}} + \text{S}_{3_{k/2}}$$

where

$\text{MUL}_2 = 4\text{AND} + 2\text{HA}$

$\text{HA} = 1\text{AND} + 1\text{XOR}$

$\text{FA} = 2\text{HA} + 1\text{XOR}$

$\text{S}_{1_m} = 2{*}(m\text{-}1)\text{FA} + 3\text{HA}$

$\text{S}_{2_m} = (2{*}m\text{-}1)\text{FA} + (m+3)\text{HA}$

$\text{S}_{3_m} = \text{FA} + (m-1)\text{HA}$

The adders required in architecture 1 (figure 6) have the cost of (4k-1)FA + (3k+5)HA. On the other hand, the adders in figure 7 have the cost of 4kFA + (2k+4)HA. While architecture 1 uses one FA less than architecture 2, the number of HA increases (k+1). Note that the latency in both architectures 1 and 2 depends on the maximum delay of the multiplier (D_{MUL}), the delay of a 2-input adder (D_{ADD_2}) and the delay of a 3-input adder (D_{ADD_3}), all them involving combinatorial logic. The performance of both architectures can be improved by inserting pipeline stages at different levels. The first pipeline level in both architectures involves inserting a register at the output of the two ($k \times k$) multipliers and also another register for the digit A_j delaying it one clock cycle in order to perform the correct operation $X_j \times Y_i + A_j$. In this case the number of cycles to compute $A^{<i+1>}$ will increase one clock cycle, leading to a latency for a Montgomery multiplication of $n \times (n + 3)$. For architecture 1, the pipeline level 2 comprises the insertion of another register at the output of the 2k-bit adders. In case of architecture 2, the pipeline level

	Arch. 1 (no pipeline)			Arch. 2 (no pipeline)				Clock cycles
k	AND	XOR	FF	AND	XOR	FF	n	(1024-bit modulus)
4	182	167	9	139	109	5	256	66048
8	754	707	17	667	589	9	128	16640
16	3074	2915	33	2899	2677	17	64	4224
32	12418	11843	65	12067	11365	33	32	1088
64	49922	47747	129	49219	46789	65	16	288

Table 4: Area and clock cycles for the *digit-digit* Montgomery multiplier for common digit sizes for k. While area complexity only depends on k, timing depends only on the modulus size and k.

2 comprises the insertion of one register at the output of the $2k$-bit adder and additionally one $2k$-bit register and one $(k+1)$-bit register for delaying R_j and $C^{<j>}$ respectively. Using a second level of pipeline causes the Montgomery multiplier to have a latency of $n \times (n+4)$ in both architectures 1 and 2.

Table 4 shows the area resources (flip-flops, AND, XOR,) and clock cycles for common operand sizes y radices in practical implementations. As it is theoretically expected, architecture 2 is slightly more compact than architecture 1, both in combinatorial resources and memory. Note than for both architectures, area and timing are inversely proportional, the greater the area resources the lesser the clock cycles. Next section discusses the implementation results of the proposed Montgomery multipliers in FPGA technology.

5. Hardware implementation and Comparisons

Architectures 1 and 2 were described in VHDL and their codes have been verified using a software reference implementation. Test vectors were used considering four common operand sizes $s \in \{128,256,512,1024,2048\}$ as well as different values for the radix $\beta = 2^k$ with $k \in \{2,4,8,16,32,64\}$. A VHDL code generator was written to produce all the posible configurations of the cartesian product $s \times k$. We selected two Xilinx FPGA devices for implementation, the Spartan xc3s500e and the Virtex5 xc5vlx50. The synthesis and place and route tool was ISE 14.2. Each architecture indicated in table 3 was implemented considering two criteria: the fist one is to implement all the architectures using the standard logic of the FPGA while the second one makes use of the in-built FPGA multiplier blocks. For all the implemented versions the memory resources for storing X, Y, p and A were mapped to the available Block Rams (BRAM) in the FPGAs, as it is usually done when scalar hardware architectures of the Montgomery algorithm are implemented. From the implementation results, an improved clock frequency is obtained for architectures with 1-level of pipeline compared with the no pipelined versions. However, for all the implementation cases the maximum path delay is in the logic for multiplication. That is, $\max\{D_{MUL}, 2D_{ADD_3}, (D_{ADD_3} + D_{ADD_2}), D_{ADD_3}\}$ $= D_{MUL}$, which depends on k. Due to this, implementation results shown in this section are only for the first four architectures listed in table 3.

Virtex5 FPGAs allow to achieve designs with higher frequencies compared to the Spartan3 device. Also, since a Virtex5 slice contains more logic (four 6-in LUTs and four flip-flops in Virtex5 and 4-in LUTs and two flip-flops in Spartan3), designs implemented in Virtex5 devices seem to be more compact. Because of this, the implementation results shown in this section consider basic building blocks such as number of LUTs and Flipflops, not in terms of slices.

Architecture 1 and Architecture 2 were implemented in four different versions. In *Version 1*, architectures 1 and 2 were implemented without pipeline stages and the multiplier blocks were implemented using the FPGA's standard logic. In *Version 2*, architectures 1 and 2 were implemented using the embedded multipliers in the FPGA. *Version 3* and *Version 4* are *Version 1* and *Version 2* respectively but implementing the pipelined version of architectures 1 and 2 (see table 3).

Figure 11 shows the implementation results of architecture 1 on the Spartan FPGA. The theoretical hardware resources shown in table 4 are confirmed in Figure 11, observing a considerably number of LUTs used for designs *Version 1* and *Version 3* when $K \geq 16$. The number of LUTs decreased considerably when the embedded multipliers are used in the designs *Version 2* and *Version 4*, being more evident for $K \geq 16$. The increasing in memory resources for the pipelined versions of architecture 1 is observed in figure 11 b). As it was stated in section 4.1.4, the greater the value of k the lesser the timing for a Montgomery multiplication. The timing shown in figure 11 c) is reduced drastically for $k \geq 16$. The pipelined versions achieve better timing, being the reduction more evident for small values of k (2,4,8,16). Figure 11 d) shows the compromise between area and performance of the implemented designs measured by the efficiency expressed as Mbps/Slice. It is concluded from figure 11 d) that the worst implementation option of architecture 1 is *Version 1* and *Version 3* which exhibit the lowest efficiency due to their high number of area resources. The better efficiency for all values of k is *Version 4*, which is the pipelined version of architecture 1 mapping the three $(k \times k)$-bit multipliers to the embedded multiplier in the Spartan3 FPGA. For all the k values considered, $k = 16$ allows to obtain the best efficiency of architecture 1 for a 1024-bit modulus. Figure 12 shows the implementation results of

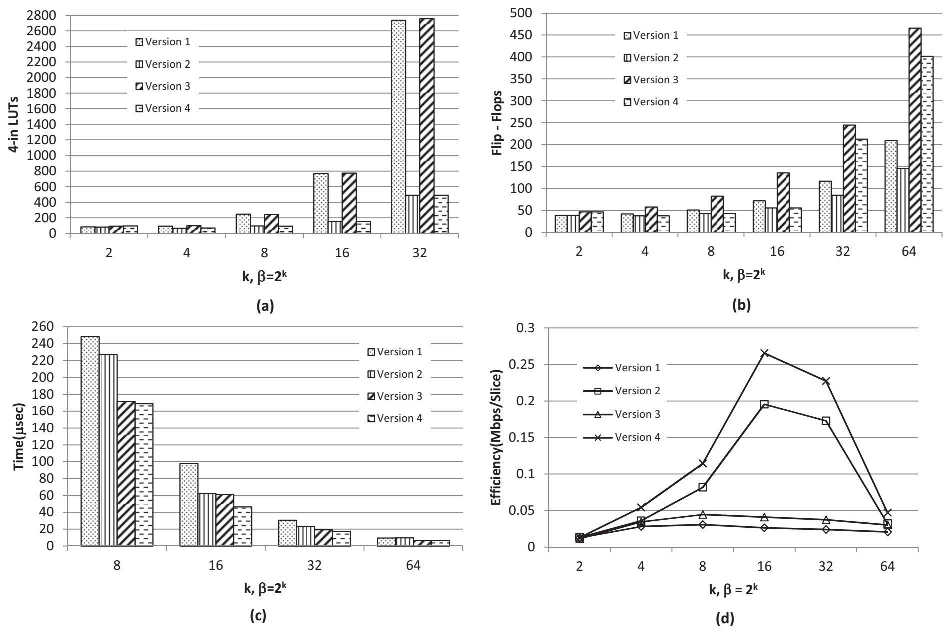

Figure 11: Implementation results for **architecture 1** in the Spartan3 FPGA device. a) Number of 4-in LUTs (combinatorial logic). b) FlipFlops used. c) Computation time of a single Montgomery multiplication for a 1024-bit modulus. d) Efficiency of the implementations expressed as Mbps/Slices (the greater the better).

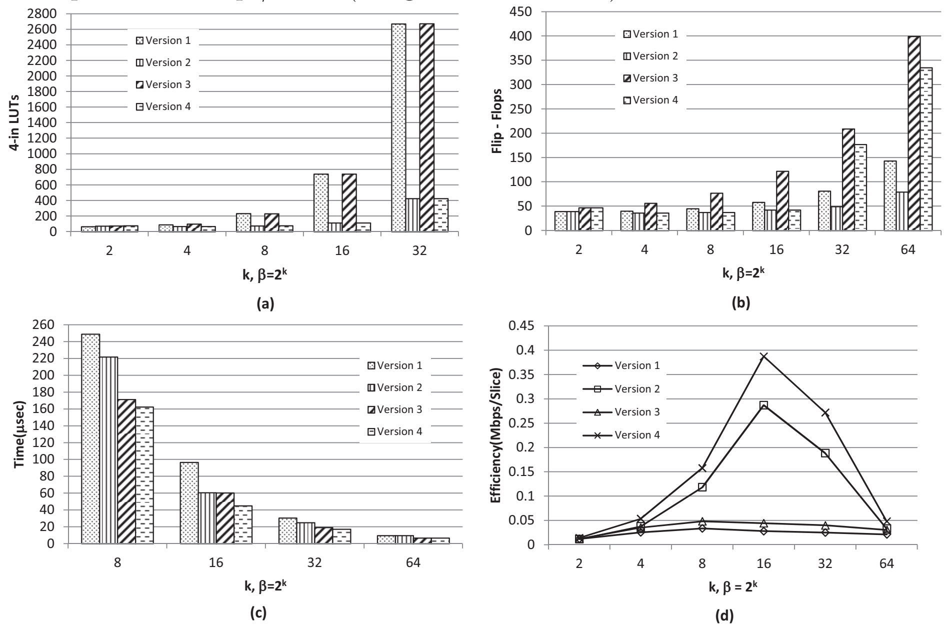

Figure 12: Implementation results for **architecture 2** in the Spartan3 FPGA device. a) Number of 4-in LUTs (combinatorial logic). b) FlipFlops used. c) Computation time of a single Montgomery multiplication for a 1024-bit modulus. d) Efficiency of the implementations expressed as Mbps/Slices (the greater the better).

architecture 2 on the Spartan FPGA using the same implementation options than architecture 1. Theoretically, architecture 2 is slightly more compact than architecture 1 (see section 4.2). This is corroborated from the results in figures 12 a) and b). In practical terms, the timing obtained by both architectures is the same but a better efficiency is achieved by architecture 2 due to its minor area consumption.

Figures 13 and 14 show the implementation results of architectures 1 and 2 in the Virtex5 FPGA. It can be observed that for *Version 1* and *Version 3*, the amount of LUTs required is practically the same than the one required in the Spartan3 implementation. However, this is not true when the in-built multiplier are used since the number of LUTs is reduced by half. By the side of memory requirements, the number of 1-bit registers needed is the same than in the Spartan3 implementations. The Virtex5 implementations lead to a better timing for computing a Montgomery multiplication.

The timing is reduced by about half but a different behavior in the results is observed respect the Spartan3 implementation. For architecture 1, *Version 2* and *Version 4* achieve almost the same time in all cases regardless *Version 4* is pipelined. This fact confirms that the maximum delay in the multiplier designs is given by the parallel $(k \times k)$-bit multiplier, being the delay by the adders in figure 6 not significant. In both architectures 1 and 2, the best timing is achieved by *Version 3*, which is a pipelined version that does not use the embedded multipliers in the FPGA. For $k \geq 16$, the timing achieved by architectures 1 and 2 is very similar. The efficiency of the hardware architectures implemented in Virtex5 is better than the one achieved in the Spartan3 implementation. The best efficiency is achieved by Montgomery

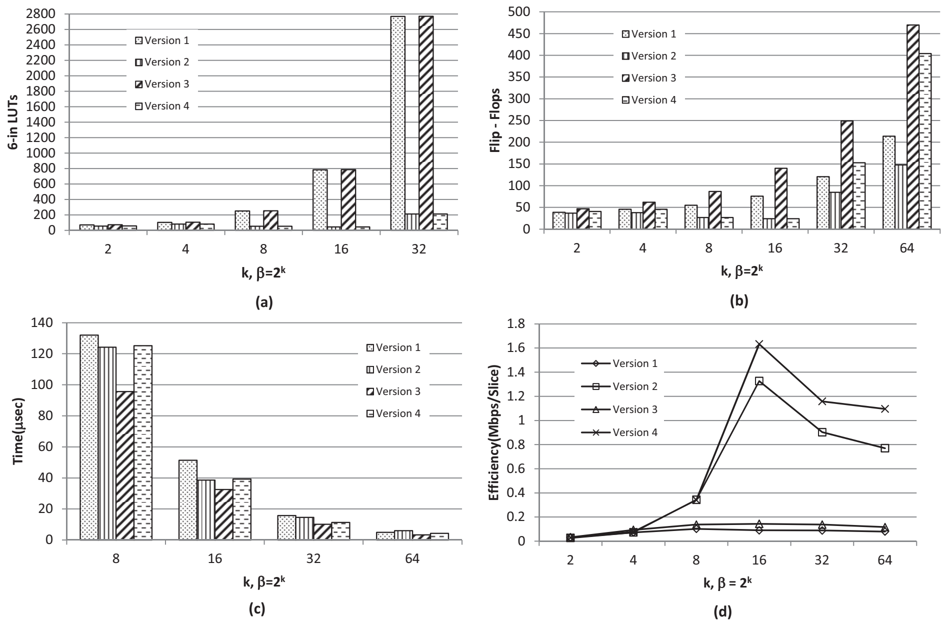

Figure 13: Implementation results for **architecture 1** in the Virtex5 FPGA device. a) Number of 6-in LUTs (combinatorial logic). b) FlipFlops used. c) Computation time of a single Montgomery multiplication for a 1024-bit modulus. d) Efficiency of the implementations expressed as Mbps/Slices (the greater the better).

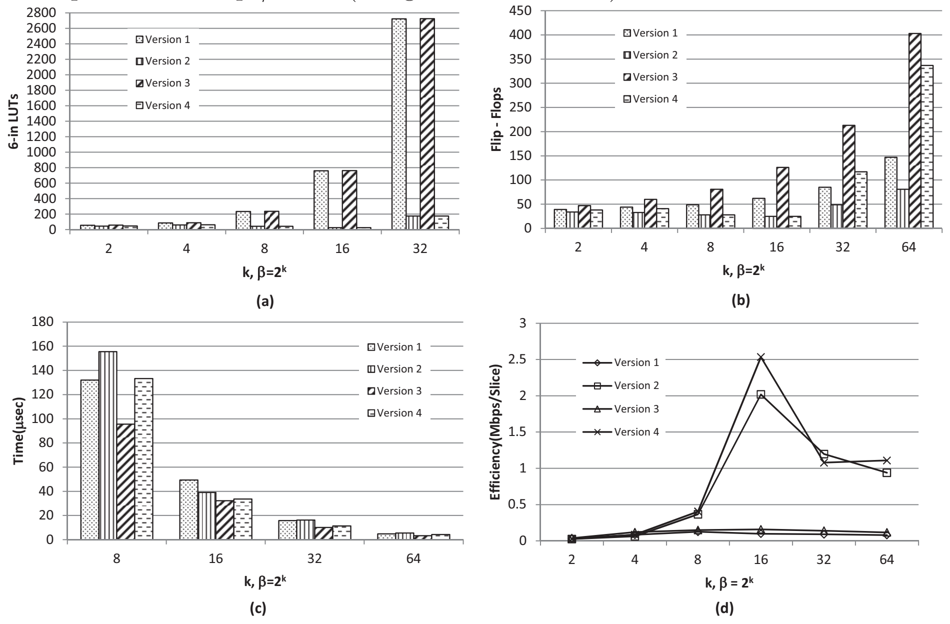

Figure 14: Implementation results for **architecture 2** in the Virtex5 FPGA device. a) Number of 6-in LUTs (combinatorial logic). b) FlipFlops used. c) Computation time of a single Montgomery multiplication for a 1024-bit modulus. d) Efficiency of the implementations expressed as Mbps/Slices (the greater the better).

Table 5: Comparison of proposed *digit-digit* Montgomery multiplier in this work against related works under similar implementation conditions.

Work	Scalable?	p (bits)	FPGA	Mults.	BRams	Slices	Time μs	Mbps	Mbps/ Slices
[24]	Yes	1020	xc3s500e	10	4	1553	7.62	133.8	0.086
This($k = 32$) Arch2-**Ver4**	Yes	1020	xc3s500e	11	4	221	18.9	60.11	0.27
[9] D-Serial(D = 4)	No	1024	Virtex5	0	-	5702	1.18	868	0.15
This($k = 64$) Arch2-**Ver3**	Yes	1024	Virtex5	0	8	2636	3.33	306.8	0.16
This($k = 64$) Arch2-**Ver4**	Yes	1024	Virtex5	33	8	219	4.21	242.66	1.10
[5] Bit-serial	Yes	512	Virtex5	0	-	957	9.64	53.11	0.05
This($k = 16$) Arch2-**Ver3**	Yes	512	Virtex5	0	4	208	8.29	61.70	0.29
This($k = 16$) Arch2-**Ver4**	Yes	512	Virtex5	4	4	10	8.67	59.02	5.90

multipliers implemented in Virtex5 using $\beta = 2^{16}$. Particularly, *Version 4* of architecture 2 is the most efficient design achieving around 2.5Mbps/Slice.

5.1. Comparisons

Comparing FPGA implementations is difficult and unfair since different FPGA technologies are considered in related works as well as different versions of synthesis tools, operand sizes, etc. The better performer GF(p) multipliers described in previous section are compared against three related works that use similar conditions to the ones considered in this paper. Table 5 show this comparison, using the same implementation devices and equal or very similar operand sizes.

From the results shown in table 5, an important fact is the significant reduced amount of area by the hardware architectures proposed in this paper compared against the results achieved in [24] and [9], where the hardware architectures follow a systolic approach. The work in [24] reports a scalable Montgomery multiplier implementing the Coarsely Integrated Operand

Scanning (CIOS) [18] Montgomery method. The work in [9] is a non scalable digit-serial Montgomery multiplier since that design is customized for full precision of operands Y and p.

For a 1024-bit modulus, our implementation of architecture 2 *Version 4* uses around 7 times less area resources than the implementation of [24] at the cost of an increasing timing around 2.5 times greater. However, our design makes better use of area resources expressed by the efficiency, which is three times better than the one achieved in [24]. Sutter et. al presented in [9] the FPGA implementation of a digit-based algorithm for Montgomery multiplication. In that design the embedded multipliers in the FPGA are not used so in table 5 we compare the results achieved in [9] with the ones obtained by architecture 2–*Version 3*, which does not make use of FPGA's multiplier embedded blocks. Again, it is observed that our design has an area reduction by half with a penalization in the timing and hence in the throughput. However, the efficiency by our multiplier is practically the same than [9], implying that both architectures process the same amount of data per area unit. The results presented in [5] are from the FPGA implementation of the bit-serial version of Montgomery algorithm. Since no multiplier blocks are used in [5], we compare those results against architecture 2–*Version 3* for $k = 16$, which achieves the best efficiency. We observe a reduction around four times in area for our design with a better computation time as well as for the efficiency. Comparing the results in [5] against the ones achieved by architecture 2–*Version 4* for $k = 16$, we observe that efficiency is improved considerably, reducing the area resources drastically but with a slight penalization in the timing.

Table 6: Performance comparison between the scalable Montgomery multiplier proposed in this work and software implementations (scalable systems).

Work	Descrip. size	Platform	Freq. (MHz)	256-bit time μs	512-bit time μs	1024-bit time μs
[10]	Software	ARM	80	-	-	570
[10]	Software	ARM	80	42.30	-	-
[25]	Software	PentiumII	400	1.57	-	-
[26]	Software	DSP	200	2.68	-	-
[27]	Software	2 MicroBlaze	100	-	139.53	539.05
[27]	Multi-core	2 MicroBlaze cores	100	-	88.04	293.82
[28]	Multi-core	4 cores 4 32x32Mults	93	2.30	-	44.00
This work	Arch2-Ver4 $k = 16$	xc3S500e	94.3	3.06	11.53	44.70
This work	Arch2-Ver4 $k = 32$	xc3S500e	63.8	1.25	4.50	17.03
This work	Arch2-Ver4 $k = 64$	xc3S500e	43.6	0.55	1.83	6.60
This work	Arch2-Ver4 $k = 16$	xc5vlx50	131.12	2.19	8.29	32.21
This work	Arch2-Ver4 $k = 32$	xc5vlx50	107.94	0.74	2.66	10.07
This work	Arch2-Ver4 $k = 64$	xc5vlx50	68.24	0.35	1.17	4.21

Our Montgomery multipliers shown in table 5 can take advantage of the embedded multipliers available un modern FPGAs. This implementation option allows to reduce drastically the area resources while keeping acceptable running times that can be tolerated in applications where area consumption is a major concern, as in applications of lightweight cryptography. The use of embedded multiplier allows to save standard logic in the FPGA that can be used to implement other modules needed by the addressed application.

Table 7: Comparison of our proposed Montgomery multiplier against other hardware implementations in the literature. N represents the operands size.

Work	Montgomery algorithm	Scalable design?	N (bits)	Platform	Area resources	Time μsec	Mbps
[9]	$R2^k MM$	No	1024	Virtex5	8227LUTs 5142FFs	1.93	530
[13] ($w = 16$)	*MWR2MM*	Yes	1024	Virtex2	9319LUTs 64DSP	9.34	109.63
[11] ($w = 16$)	*MWR2MM*	Yes	1024	Virtex2	4178slices 65PEs	10.88	94.11
[10] ($w = 16$)	*MWR2MM*	Yes	1024	0.5μm CMOS	28KGates	43	23.81
This Arch2-Ver4 ($k = 32$)	Proposed *IDDMM*	Yes	1024	Spartan3	425LUTs 177FFs 11 18x18Mults.	17.03	60.23
This Arch2-Ver4 ($k = 64$)	Proposed *IDDMM*	Yes	1024	Virtex5	704LUTs 337FFs 33DSP48E	4.21	242.66
[6] Fully-parallel	Original	No	256	xchvhx250T	2106 Slices 16DSP48E	0.014	18285.71
[29] Fully-parallel	Original	No	256	Cyclone3	23405 LEs 81Mults	0.1	2560
This Arch2-Ver4 ($k = 32$)	Proposed *IDDMM*	Yes	256	Spartan3	419LUTs 177FFs 11 18x18Mults.	1.25	204.37
This Arch2-Ver4 ($k = 64$)	Proposed *IDDMM*	Yes	256	Virtex5	699LUTs 333FFs 33DSP48E.	0.35	727.90

Table 6 shows a comparison of the proposed scalable Montgomery multiplier against other scalable system (software based) that use general purpose microprocessors [10, 25, 26] with similar clock frequencies. That comparison allows to justify the motivation to perform Montgomery multipliers as dedicated hardware modules. In most of the cases, the results achieved by our proposed Montgomery multiplier outperform related works, inclusive for parallel implementation as the multi-core implementations in [28] and [27].

Finally, in order to compare our Montgomery multiplier against other hardware implementation approaches, we provide a comparison in table 7 only as a reference. Some works mentioned in section 1 as related works were not included in this table since they do not provide implementation results (slices, LUTs, timing), for example [12] and [14]. Table 7 reveals the big amount of area resources required in related words regardless the technology used. In most of the cases, our proposed multiplier uses the least amount of area resources with a penalization in the computation time that could be tolerated in practical applications where area consumption is a major concern.

6. Conclusions

Modular multiplication in GF(p) is one of the most time consuming operations in practical implementations of public key cryptosystems, such as RSA and Elliptic Curve Cryptography. Although several work has been done in the literature proposing hardware architectures that speed up modular multiplication using the Montgomery method, most of those works have focused in achieving high performer solutions. That design goal in previous works has lead to hardware architectures requiring great amount of area resources, that in case of FPGA implementations that means great amount of slices (LUTs and flip-flops) used and thus, the occupation of most of the device's area resources. However, practical implementations of public cryptographic schemes involve more operations not only modular multiplication. Saving area resources in the FPGA is very attractive because more standard logic is available for implementing other parts of the cryptographic application. For example, more logic would be available for implementing the exponentiation module needed for encryption/decryption in RSA.

The novel *IDDMM* algorithm for Montgomery multiplication based on the *digit-digit* computation presented in this paper and its corresponding hardware implementation performs better than software counterparts and at the same time, that implementation requires fewer area resources than other related works, making use of available multipliers embedded in modern FPGAs. Although the performance is reduced compared to other hardware implementation approaches, that penalization is tolerable since the timing is still competitive and it could be tolerated in practical cryptographic applications.

Area and time complexity of the proposed hardware architectures for the *IDDMM* algorithm depends mainly on the radix $\beta = 2^k$ used. That allows to count with a scalable multiplier that use the same hardware for computing modular multiplications independently of operands size. The different implementation options studied in this paper allows to establish area-performance trade-offs. From the experimental results, the most efficient multiplier results using $\beta = 2^{16}$ as radix. Although the use of in-built multipliers reduces considerably the area resources, the performance is slightly decreased. For all cases studied, the maximum critical path delay in the designs is determined by the $(k \times k)$ multipliers. The Montgomery multiplier described in this work is well suited to serve a the main core of an exponentiation circuitry for RSA encryption.

References

[1] R. L. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, Commun. ACM 21 (2) (1978) 120–126.

[2] N. Koblitz, Elliptic Curve Cryptosystems, Mathematics of Computation 48 (177) (1987) 203–209.

[3] V. Miller, Use of Elliptic Curves in Cryptography, in: Proc. of Advances in Cryptology, CRYPTO'85, Santa Barbara, CA, August 1985, pp. 417–426.

[4] P. L. Montgomery, Modular multiplication without trial division, Math. Computation 44 (1985) 519–521.

[5] M. Hamilton, W. Marnane, A. Tisserand, A comparison on FPGA of modular multipliers suitable for elliptic curve cryptography over GF(p) for specific p values, in: 2011 International Conference on Field Programmable Logic and Applications (FPL), 2011, pp. 273 –276. doi:10.1109/FPL.2011.55.

[6] A. Mondal, S. Ghosh, A. Das, D. R. Chowdhury, Efficient FPGA implementation of Montgomery multiplier using DSP blocks, in: Proceedings of the 16th international conference on Progress in VLSI Design and Test, VDAT'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 370–372. doi:10.1007/978-3-642-31494-0_47.

[7] G. C. T. Chow, K. Eguro, W. Luk, P. Leong, A Karatsuba-based Montgomery multiplier, in: Proceedings of the 2010 International Conference on Field Programmable Logic and Applications, FPL '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 434–437. doi:10.1109/FPL.2010.89.
URL `http://dx.doi.org/10.1109/FPL.2010.89`

[8] A. F. Tenca, Çetin Kaya Koç, A scalable architecture for Montgomery multiplication, in: Workshop on Cryptographic Hardware and Embedded Systems, 1999, pp. 94–108.

[9] G. Sutter, J.-P. Deschamps, J. L. Imaña, Modular multiplication and exponentiation architectures for fast RSA cryptosystem based on digit serial computation, IEEE Transactions on Industrial Electronics 58 (7) (2011) 3101–3109.

[10] A. F. Tenca, Çetin Kaya Koç, A scalable architecture for modular multiplication based on Montgomery's algorithm, IEEE Trans. Computers 52 (9) (2003) 1215–1221.

[11] M. Huang, K. Gaj, S. Kwon, T. El-Ghazawi, An optimized hardware architecture for the Montgomery multiplication algorithm, in: Proceedings of the Practice and Theory in Public Key Cryptography, 11th International Conference on Public Key Cryptography, PKC'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 214–228.

[12] Y. Kong, High radix Montgomery multipliers for residue arithmetic channels on FPGAs, in: D. Zeng (Ed.), Future Intelligent Information Systems, Vol. 86 of Lecture Notes in Electrical Engineering, Springer Berlin Heidelberg, 2011, pp. 23–30. doi:10.1007/978-3-642-19706-2_4.

[13] M. Huang, K. Gaj, T. A. El-Ghazawi, New hardware architectures for Montgomery modular multiplication algorithm, IEEE Trans. Computers 60 (7) (2011) 923–936.

[14] A. F. Tenca, G. Todorov, Çetin Kaya Koç, High-radix design of a scalable modular multiplier, in: Workshop on Cryptographic Hardware and Embedded Systems, 2001, pp. 185–201.

[15] C. D. Walter, Montgomery's multiplication technique: How to make it smaller and faster, in: Workshop on Cryptographic Hardware and Embedded Systems, 1999, pp. 80–93.

[16] M. Morales-Sandoval, A. Diaz-Perez, A compact FPGA-based Montgomery multiplier over prime fields, in: Proceedings of the 23rd ACM International Conference on Great Lakes Symposium on VLSI, GLSVLSI '13, ACM, New York, NY, USA, 2013, pp. 245–250. doi:10.1145/2483028.2483102.

[17] A. A. Ibrahim, H. A. Elsimary, A. M. Nassar, FPGA design, implementation and analysis of scalable low power radix 4 Montgomery multiplication algorithm, WSEAS Trans. Cir. and Sys. 6 (12) (2007) 601–607.
URL `http://dl.acm.org/citation.cfm?id=1486868.1486869`

[18] C. K. Koç, T. Acar, B. S. Kaliski, Jr., Analyzing and comparing Montgomery multiplication algorithms, IEEE Micro 16 (3) (1996) 26–33. doi:10.1109/40.502403.
URL `http://dx.doi.org/10.1109/40.502403`

[19] C. Walter, Montgomery exponentiation needs no final subtractions, Electronics Letters 35 (21) (1999) 1831–1832. doi:10.1049/el:19991230.

[20] N. Mentens, K. Sakiyama, B. Preneel, I. Verbauwhede, Efficient pipelining for modular multiplication architectures in prime fields,

in: Proceedings of the 17th ACM Great Lakes symposium on VLSI, GLSVLSI '07, ACM, New York, NY, USA, 2007, pp. 534–539. doi:10.1145/1228784.1228911.
URL `http://doi.acm.org/10.1145/1228784.1228911`

[21] Altera Corporation, Implementing multipliers in FPGA devices. Application note 306 ver. 3.0, 2004, pp. 1–48.
URL `http://www.altera.com/literature/an/an306.pdf`

[22] Y. Zhanga, G. Shoub, Y. Huc, Z. Guo, Low complexity GF(2^m) multiplier based on iterative Karatsuba algorithm, Advanced Materials Research (2012) 1409–1414.

[23] E. Cuevas-Farfan, M. Morales-Sandoval, A. Morales-Reyes, C. Feregrino-Uribe, I. Algredo-Badillo, P. Kitsos, R. Cumplido, Karatsuba-Ofman Multiplier with Integrated Modular Reduction for GF(2^m), Advances in Electrical and Computer Engineering 13 (2) (2013) 3–10. doi:10.4316/aece.2013.02001.
URL `http://dx.doi.org/10.4316/aece.2013.02001`

[24] E. Oksuzoglu, E. Savas, Parametric, secure and compact implementation of RSA on FPGA, in: International Conference on Reconfigurable Computing and FPGAs, 2008, pp. 391 –396. doi:10.1109/ReConFig.2008.13.

[25] M. Brown, D. Hankerson, J. López, A. Menezes, Software implementation of the NIST elliptic curves over prime fields, in: Proceedings of the 2001 Conference on Topics in Cryptology: The Cryptographer's Track at RSA, CT-RSA 2001, Springer-Verlag, London, UK, UK, 2001, pp.

250–265.
URL `http://dl.acm.org/citation.cfm?id=646139.680803`

[26] K. Itoh, M. Takenaka, N. Torii, S. Temma, Y. Kurihara, Fast implementation of public-key cryptography on a DSP TMS320C6201, in: Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems, CHES '99, Springer-Verlag, London, UK, UK, 1999, pp. 61–72.
URL `http://dl.acm.org/citation.cfm?id=648252.752380`

[27] Z. Chen, P. Schaumont, A parallel implementation of Montgomery multiplication on multicore systems: Algorithm, analysis, and prototype, IEEE Transactions on Computers 60 (12) (2011) 1692–1703. doi:10.1109/TC.2010.256.

[28] J. Fan, K. Sakiyama, I. Verbauwhede, Montgomery modular multiplication algorithm on multi-core systems, in: 2007 IEEE Workshop on Signal Processing Systems, 2007, pp. 261–266. doi:10.1109/SIPS.2007.4387555.

[29] Y. Gong, S. Li, High-throughput FPGA implementation of 256-bit Montgomery modular multiplier, in: IEEE Second International Workshop on Education Technology and Computer Science, 2010, pp. 173 –177.

Appendix A. Test Vectors

All the test vectors presented in this section were generated and validated from a software implementation.

1024-bit operands (RSA modulus, radix-16 notation)

$p =$ e603bcf9fa9b405cd851ac0a3d33f9120c8957e79825c2a5bdae35000c5e6b1d
302162200dd35659c2ae138eff1e6bb394a745f0f871b8af861371106fa0db08
7c74ac64df7c8b41f3363f7a791d833d680290523fc74d0b99260744681bfe8c
c70b677d15d1546a34f2f4d361a43fed28555239471420e41a82e74d576982cf

$Y =$ 7481eb9b0d819f15632afd73befe15ece799cf7cd1893cc89c606226c4d1d42d
26db5a67ddd5ec1966e0f835c3bfc091b94ddcbbe85d53e2e7fe81e94f7e0c31
addb904c7af1630904bbb71b80f62e45977cacbf9e8c868e04f3471ec1733a2a
d99dfbe5d6d4ad8573d09a79e76b7779234bfce84ff12324488e39d285345bed

$X =$ 22ff5b47eb95c38b0a888e57377998711376cda83d693af54f7944012c282577
4530af1b597b9b905653095ebc3ca3ef2e451934cebd839662a2e58d41d8c446
2966e56d5af09f615530eab1fa857d09f9b80e649d0c0c94559884f2d76fdae7
0dcea6640f39912aacf016ea938c68b005372d1b3477dbeef0ae4530081b8724

p'	k
1	2
1	4
d1	8
bbd1	16
7a81bbd1	32
f95fe1807a81bbd1	64

$\mathrm{MM}(X, Y, p) =$ e79f12a5abfe489e6f272db5620ecd084d0f85cde73b3588b4271e21908cfd66
c5df8a669f73cfc09955edba71dc9b7da72ab9bf84f13426a3d65a36b8b5722e
62b850032384fc87e555df0d9449f16333560ae8bc54dc7cc56934c2a45e8c16
918bff26bb3af459ebcc01dca17bdb620896b391e7d3ec5519f276cb7e8f7faf

512-bit operands (RSA modulus, radix-16 notation)

$p =$ eaa0f7b011d858bc1fe7d9eae62be36848397a0c165de35895dbb7cbe8f024b4
65625aeb2808790a305318c53635dc5cf6667744f2b4ba46cf300adf05ae4023

$Y =$ 7c9cc98a468c3be0469dabefd0a32bfb3ccb9cfe9d43a8ac05f482b9a1464b18
2a26b9ebdf722ed50a701659fe1b870527875307d67691c0edb251376b907570

$X =$ 487e997d6e9e5d2b19c4ffe6302defcaa0cbfe8a0b465f113baf7a0ebb29ed46
e64f2cb1550880b825e913e8386623a6a9a6c3a1e7e4fb9c81307616399ea35

p'	k
1	2
5	4
75	8
9075	16
8fa39075	32
cbc9bf1e8fa39075	64

MM$(X, Y, p) =$ a817b263c6ac8ecb920efa19d6a724faea93d513369087503c7683b880aee172
1100757f0ddc441d5e6b5b491c8227e8e8e360867752924c9f50e4ecc0fcd5d

256-bit operands (ECC modulus secp256k1, radix-16 notation)

$p =$ fffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f

$Y =$ de6501bd55b07ce9c83bbcbba280e5700e53c152304f6a1ab183a7b2e16e308

$X =$ 6113b410fe0d6b547a64ce68e9b7430214e56ec57e37d50dc22be4fe5e5f8d2f

p'	k

1	2
1	4
31	8
3531	16
d2253531	32
d838091dd2253531	64

$\mathrm{MM}(X, Y, p) =$ 80b139fdb1400ce5fbf1ddcfde81e294dfdb046b0e306ac7d65c6a707344c744