Low Power Montgomery Modular Multiplication on Reconfigurable Systems

Pedro Maat C. Massolino[1], Lejla Batina[1], Ricardo Chaves[2], and Nele Mentens[3]

[1] Radboud University, ICIS - Digital Security Group, The Netherlands
Email:{pmassolino,lejla}@cs.ru.nl
[2] Department of Electrical and Computer Engineering, IST INESC-ID, Lisbon - Portugal
Email:ricardo.chaves@inesc-id.pt
[3] ESAT/COSIC, Katholieke Universiteit Leuven, Belgium
Email:nele.mentens@esat.kuleuven.be

Abstract. This paper presents an area-optimized FPGA architecture of the Montgomery modular multiplication algorithm on a low power reconfigurable IGLOO® 2 FPGA of Microsemi®. Our contributions consist of the mapping of the Montgomery algorithm to the specific architecture of the target FPGA, using the pipelined Math blocks and the embedded memory blocks. We minimize the occupation of these blocks as well as the usage of the regular FPGA cells (LUT4 and Flip Flops) through an dedicated scheduling algorithm. The obtained results suggest that a 224-bit modular multiplication can be computed in 2.42 μs, at a cost of 444 LUT4, 160 Flip Flops, 1 Math Block and 1 64x18 RAM, with a power consumption of 25.35 mW. If more area resources are considered, modular multiplication can be performed in 1.30 μs at a cost of 658 LUT4, 268 Flip Flops, 2 Math Blocks, 2 64x18 RAMs and a power consumption of 36.02 mW.

1 Introduction

Electronics advancements in the last decades have made communication systems available to everyone. However, to assure the secure transmission of sensitive information through public channels, cryptographic primitives need to be employed. Given the costly computation of these primitives, in particular asymmetric encryption algorithms, hardware implementations are often necessary. Particularly on embedded systems, hardware implementations allow to achieve more efficient computations with a lower power consumptions.

When creating a new application, system engineers need to choose an appropriate public key primitive. To guide on this choice some public key standards have been made [1, 14, 16, 25]. With this broad arrange of different standards, each one being applied/enforced by different countries or application scenarios, it is necessary to provide implementations that are sufficiently versatile to accommodate them all. Examples of standardized public key cryptosystems are based on RSA [28] and elliptic curve cryptography (ECC) [22]. Most standards based on RSA [16, 25] are compatible with each other, which makes it possible to use an RSA implementation for various purposes. For ECC [12, 14, 25], the different flavors of curve parameters and finite fields make the implementations less versatile than RSA implementations. Some standards, such as NIST-186 [24] are based on Solinas primes [31], which allow for an efficient modular reduction algorithm. On the other hand, primes suggested by Brainpool [14] are just random primes with no special arithmetic characteristic. Therefore, they only work with generic reduction algorithms. Newer ECC proposals [3, 6] are using Crandall primes [9], which also have a specific algorithm for modular reduction. Even with all those differences, all standard algorithms based on ECC and RSA share the same mathematical basis: modular multiplication of large integers.

Modular multiplication of large integers, can be efficiently implemented through the Montgomery multiplication algorithm [23]. This algorithm can be applied for RSA and ECC with prime fields operations. Particularly in ECC, it is possible to give support for all primes, even those with no special modular reduction algorithms. Montgomery multiplication avoids the division operation, which is difficult to implement in an efficient way in hardware. Barrett multiplication [4] achieves the same goal. Nevertheless, we choose to focus this paper on Montgomery multiplication, since it allows to avoid modular addition and subtraction reduction operations [5,37], those reductions are operand dependent and therefore introduce a possible side-channel leakage. Besides, if the reductions were done it would have the same or more cost than avoiding them, therefore it is preferable to avoid, given it is cost.

We propose two compact hardware co-processors for Montgomery multiplication and integer addition/subtraction. With these operations available it is possible to implement an ECC co-processor, which can be used for all ECC parameters and prime fields. Both architectures strive for a low area footprint and low power consumption, but the second one deploys more resources in order to provide a better performance.

To implement Montgomery multiplication in scenarios where the inputs are stored and computed as multiple small words, Koç et al. [18] analyzed 5 algorithms: Separated Operand Scanning (SOS), Coarsely Integrated Operand Scanning(CIOS), Finely Integrated Operand Scanning (FIOS), Finely Integrated Product Scanning (FIPS) and Coarsely Integrated Hybrid Scanning(CIHS). All algorithms were analyzed according to memory space, number of arithmetic operations and number of memory access. When comparing in respect with multiplications, all 5 algorithms have the same amount. However, SOS requires almost twice the amount of memory than the others, because it computes the entire product before applying modular reduction. Given it is higher memory requirement it was opted out for not being implemented. From those remaining it is possible to split them into two groups: operands scanning and product scanning. CIOS and FIOS apply operand scanning, which means all operations are done while scanning both inputs, which is straightforward to mapping into hardware. Product scanning on the other hand requires more control logic, and complex addressing system, because both inputs are scanning with one being incremented and the latter decremented. Given a more complex control logic and addressing system, both FIPS and CIHS were discarded for this implementation.

In the literature there is a great number of co-processors for modular multiplication with the Montgomery algorithm [2, 7, 8, 13, 15, 20, 30, 34, 36]. All proposals show results for modular multiplication, some also give implementation results for public key cryptosystems like RSA and ECC. For our implementation we give results for circuit area, execution time and power simulations.

Our first implementation aims at a IGLOO® 2 [21] FPGA from Microsemi®. This FPGA model is based on FLASH technology as opposed to SRAM technology used in Xilinx® and Altera® FPGAs. Since it is based on a different technology it is labeled as more energy efficient [21]. With a energy efficient platform and a area optimized strategy it is expected to have low power results.

The remainder of this article is divided in the following sections. Section 2 gives the necessary background information on Montgomery multiplication and the FPGA family chosen for this study. Section 3 shows our two hardware architectures and the scheduling of the resources. Section 4 compares our results with the literature. Finally, Section 5 gives some final considerations and future works.

2 Background

In the first part of this section we discuss the Montgomery algorithm and two word based level variants, CIOS and FIOS algorithms [18]. In the second part we give details on the FPGA applied in this work, IGLOO® 2 from Microsemi®.

2.1 Montgomery algorithm

Modular multiplication of large integers is essential for various cryptosystems [11, 22, 28], but if implemented naively it does not fulfill the performance requirements of certain applications. Montgomery [23] discovered that computing $a \cdot b/r \bmod n$, where r is a power of 2 greater than n, with n being a non-multiple of 2, is more efficient than computing $a \cdot b \bmod n$ directly. This efficiency results from increasing each partial product by a multiple of n and then dividing by r, being less expensive than computing the modular reduction through a division by n. Montgomery multiplication, shown in Algorithm 1, first computes the product p of a and b. Then it multiplies the result by n', which gives m, a number that when multiplied by n and added to p, makes the result a multiple of r. This multiple is divided r through a shift operation.

In the original Montgomery algorithm [23], after the division by r there is an extra reduction step. In this extra step, n is subtracted from p if p is bigger than n. This extra step can lead to side-channel leakages [19]. To counter this problem, Walter proposed to increase r by at least two extra bits, thus eliminating the need for the final subtraction [37].

The inputs a and b of Algorithm 1 should be less than or equal to $2n$. If r is expanded by 4 bits, instead of only 2, inputs a and b can be up to $4n$ [5].

Algorithm 1 Montgomery multiplication algorithm [23], without final subtraction [37]

```
Require: $a, b \leq 2n$, $r = 2^{\lceil log_2(n) \rceil + 2}$, $r \cdot r^{-1} - n' \cdot n = 1$
Ensure: $o = a \cdot b/r \pmod n$
 1: $p \leftarrow a \cdot b$
 2: $m \leftarrow n' \cdot p \pmod r$
 3: $o \leftarrow (p + m \cdot n)/r$
 4: return $o$
```

Therefore, in case a and b are less than n, it is possible to add and/or subtract another variable less than n to a and b before multiplication. By increasing the number of bits even more it is possible to avoid two or more consecutive additions or subtractions, thus allowing for the optimization of ECC formula.

In the CIOS algorithm, described in Algorithm 2, the partial products of the entire value a times, a particular word of b, are computed first, and then m is computed, just as in Algorithm 1. In this case m has the size of one word as opposed to the double operand size in Algorithm 1. As consequence, n' also has the same size as m. After computing the partial product and m, a reduction is applied on word-size variables. This process is repeated in the next partial product computation. CIOS works by exploiting the fact that each partial product needs a different number of zeroes is added, before reduction.

In the FIOS algorithm, shown in Algorithm 3, instead of computing each partial product and then applying the modular reduction, as CIOS, both operations occur at the same time. In the first iteration step, after computing $a_0 \cdot b_i$, the result is multiplied by n', resulting in m. Then, a new iteration computes a_j times b_i and adds the result to m times n_j. This process is repeated for all words of a and b. Because both partial product computation and reduction are done at the same loop, FIOS have to access more memory positions than CIOS. Hence, this extra memory access can insert wait states on the architecture, so for such architectures CIOS or other algorithms are preferable.

2.2 IGLOO® 2 FPGAs

The target FPGA, IGLOO® 2 [21] from Microsemi®, is a flash based architecture with some specialized block like memories and Math Blocks. The Math Blocks behave like Digital Signal Processors (DSPs) in the sense that they can do some multiplications in combination with additions and accumulations. These Math Blocks have a total of 3 inputs available to the user, two operands for multiplication and one operand for addition with the multiplication result (it is also possible to negate the multiplication result). It is also possible to add the result, or have the result shifted to the right by 17 bits. Another option includes a special lane to connect the input of one Math Block with the output of a previous one. This dedicated lane allows for a fast result propagation, producing an efficient way to process larger operands.

Algorithm 2 Word based Montgomery multiplication algorithm (CIOS) [18], without final subtraction [37]

```
Require: $a, b \leq 2n$, $r = 2^{(\lceil(\lceil log_2(n)+4\rceil/\text{word size})\rceil\cdot\text{word size})}$, $r \cdot r^{-1} - n' \cdot n = 1$, $w = 2^{\text{word size}}$, $l = \lceil log_2(r/w)\rceil$
Ensure: $o = a \cdot b/r \pmod n$
 1: for $i \leftarrow 0$ to $l-1$ by $1$ do
 2:     $c \leftarrow 0$
 3:     for $j \leftarrow 0$ to $l-1$ by $1$ do
 4:         $t \leftarrow o - j + a_j \cdot b_i + c$
 5:         $p_j \leftarrow t \pmod w$, $c \leftarrow t/w$
 6:     end for
 7:     $t \leftarrow o_l + c$
 8:     $p_l \leftarrow t \pmod w$
 9:     $p_{l+1} \leftarrow t/w$
10:     $m \leftarrow n' \cdot p_0 \pmod w$
11:     $c \leftarrow (p_0 + m \cdot n_0)/w$
12:     for $j \leftarrow 1$ to $l-1$ by $1$ do
13:         $t \leftarrow p_j + m \cdot n_j + c$
14:         $o_{j-1} \leftarrow t \pmod w$, $c \leftarrow t/w$
15:     end for
16:     $t \leftarrow o_l + c$
17:     $o_{l-1} \leftarrow t \pmod w$
18:     $o_l \leftarrow t/w$
19: end for
20: return $o$
```

Algorithm 3 Word based Montgomery multiplication algorithm (FIOS) [18], without final subtraction [37]

```
Require: $a, b \leq 2n$, $r = 2^{(\lceil(\lceil log_2(n)+4\rceil/\text{word size})\rceil\cdot\text{word size})}$, $r \cdot r^{-1} - n' \cdot n = 1$, $w = 2^{\text{word size}}$, $l = \lceil log_2(r/w)\rceil$
Ensure: $o = a \cdot b/r \pmod n$
 1: for $i \leftarrow 0$ to $l-1$ by 1 do
 2:     $t \leftarrow o_0 + a_0 \cdot b_i$
 3:     $p \leftarrow t \pmod w$, $c1 \leftarrow t/w$
 4:     $m \leftarrow n' \cdot p \pmod w$
 5:     $c2 \leftarrow (p + m \cdot n_0)/w$
 6:     for $j \leftarrow 1$ to $l-1$ by 1 do
 7:         $t \leftarrow o_j + a_j \cdot b_i + c1$
 8:         $p \leftarrow t \pmod w$, $c1 \leftarrow t/w$
 9:         $t \leftarrow p + m \cdot n[j] + c2$
10:         $o_{j-1} \leftarrow t \pmod w$, $c2 \leftarrow t/w$
11:     end for
12:     $t \leftarrow o_l + c1$
13:     $p \leftarrow t \pmod w$, $c1 \leftarrow t/w$
14:     $t \leftarrow p + c2$
15:     $o_{l-1} \leftarrow t \pmod w$, $c2 \leftarrow t/w$
16:     $o_l \leftarrow c1 + c2$
17: end for
18: return $o$
```

This FPGA also includes Dual-Port Large SRAM with two inputs and two outputs. It can do 2 reads, 1 read and 1 write or 2 writes at the same time. Both ports behave independently and share the same memory region. This memory can be configured to 1024 words of 18 bits.

Another available memory to work with is the Three-Port Micro SRAM, which has 2 read outputs and 1 write input. With this memory it is possible to do 2 reads and 1 write at the same time. However, this memory can only store 64 words of 18 bits. As a side note, this memory does not have a collision system in case the same position is being read and written at the same time. For this reason, loads and stores on the memory need to be carefully designed.

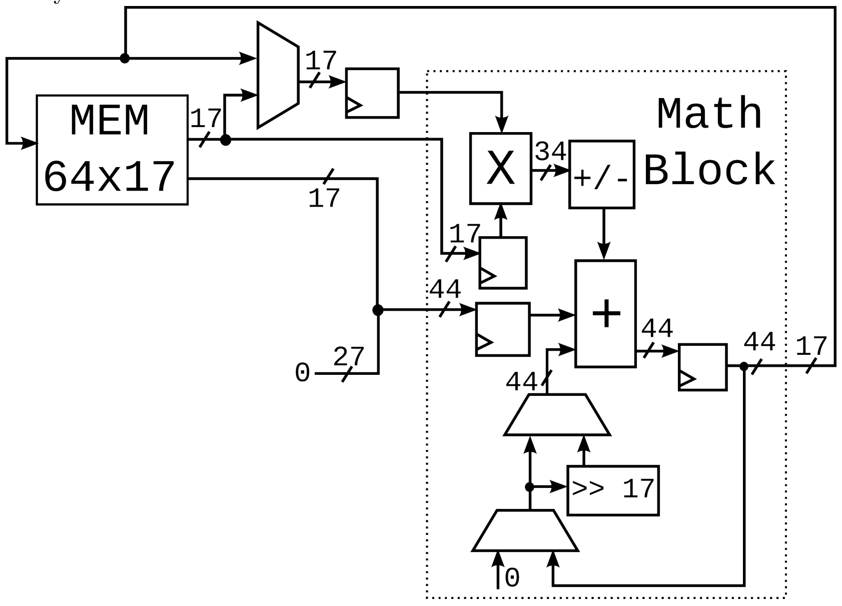
3 Implementation Considerations and Proposed Structure

Since the CIOS and FIOS algorithm are word-based, either the size of r needs to be increased to a multiple of the word size or the last round of the algorithms needs to be adapted to work on variables with a size smaller than the word size. To minimize the control logic and to maximize the re-use of resources, we opted for increasing the size of r to be a multiple of the word size. This comes with a negligible decrease of speed.

In the first proposed architecture, we try to minimize the number of Math Blocks and the number of memories to 1, as depicted in Figure 1. Because of the internal memory size, it is only possible to store the multiplication operands. The Math Block has internal registers for inputs and outputs which can be leveraged, however for one input it was necessary to add an external one. This extra register is necessary for applying subtraction operations.

To decrease the response time, another architecture, Figure 2, is proposed. In this architecture it was opted to increase the number of Math Blocks and Block RAMs, to only 2. With two, it is possible to compute the partial product in one Math Block and apply the reduction in the second. However, for both operations to work properly it is necessary to have two memories to provide the necessary values. In this architecture, 2 extra registers were added, one for each Math Block. Just as the previous architecture this is to support the subtraction operation.

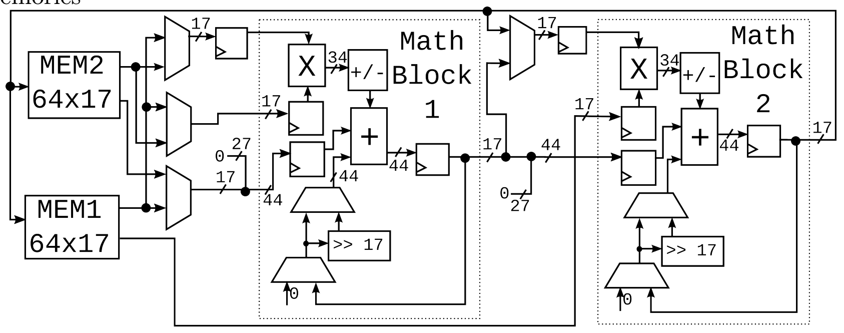
Fig. 1. Architecture version 1 of the Montgomery multiplier using 1 Math Block and 1 memory



For the control system, some address registers were added to indicate the location of each variable in the memory, except for the variable n which is always at the first location in the memory and n' followed by n. These registers in the architecture allow to use the output of an operation directly as the input of the next operation.

To get minimum latency, the system leverages the internal Math Block as much as possible. All operations follow each other closely, so during a multiplication both the Math Block and the memory are working. The Math Block works in a pipeline mode, loading values for next computation, while processing the

Fig. 2. Architecture version 2 of the Montgomery multiplier using 2 Math Blocks and 2 memories



current computation and writing the result of the previous computation to the memory.

The sequence of execution of Algorithm 2 on the architecture in Figure 1 is given in Table 1. In the beginning, the first word of b is loaded. In the next cycle, the first word of a is loaded and the Math Block registers $RegC$ and $RegP$ are reset to 0. Then with all three ready, Math Block register $RegP$ receives $RegA \cdot RegB + RegC + (RegP << 17)$, which explains the reason Math Block register $RegP$ and $RegC$ are reset. This process is repeated by loading the next word of a and computing $RegP = RegA \cdot RegB + RegC + (RegP << 17)$. The shifted value of $RegP$ is the carry out of the previous computation. Therefore, the complete process computes $a \cdot b_i$.

After loading the last word of a, the process to compute m begins, while the last results of the partial product are computed. First, $RegA$ is loaded with n' and $RegB$ is reset to 0. This reset occurs so the multiplication of $RegA$ and $RegB$ does not interfere with the last partial product. Then, one cycle is lost because the Math Block needs to wait for the partial product p_{size+1} computation. In the next cycle, p_0 is loaded into $RegB$ and then value m is computed by $RegP = RegA \cdot RegB + RegC$, and has to wait one extra cycle to be loaded into $RegA$. This extra wait is necessary because it is not possible to do a direct connect to $RegA$. However, it is possible to add an extra multiplexer after $RegA$ output. This approach was not chosen because it would decrease the operating frequency, thus resulting in an efficiency loss, and introduce extra hardware resources for the multiplexer.

After m is computed, words n_0 and p_0 are loaded into register $RegB$ and $RegC$, also $RegP$ is reset. After loading all values, $RegP = RegA \cdot RegB + RegC + (RegP << 17)$ is computed and the result is kept in $RegP$ for carry purposes on the next values n_1 and p_1, but is not stored in memory. Then the computation with n_1 and p_1 is stored in memory as o_0. The process is repeated for all words in n and p. In the last stages both $RegA$ and $RegB$ are reset, so in the next cycle $RegB$ can receive a_0 without interrupting the computation of

o_{size} . At the next cycle, word b_1 and p_0 are loaded and the process can continue until all b_i words have been processed.

The architecture in Figure 2 executes Algorithm 3 as seen in Table 2. In the beginning register *RegB1* receives b_0, and in the next cycle *RegC1* is reset and *RegA1* receives a_0. With those values, $RegP1 = RegA1 \cdot RegB1 + RegC1$ is computed and the first partial product is computed. Unlike Algorithm 2, in this algorithm the value m is computed after computing the first partial product, so Math Block 2 can apply the reductions. In the next cycle, p_0 is loaded into register *RegM2* and n' is loaded into *RegB2*, also *RegC2* is reset. After all values loaded, value m is computed and loaded into register *RegM2*. To reduce waiting cycles, when m is computed, register *RegA1* receives the next word a_1 to continue the partial product computation. When m is in register *RegM2*, register *RegB2* is loaded with n_0 and *RegC2* with p_0. The process continues with Math Block 1 generating the partial product and Math Block 2 applying the Montgomery reduction.

This process continues until all words of a have been computed, then register *RegA1* and *RegB1* are reset. After resetting it waits for one cycle, then begins loading the next word b_1 into *RegB1*, and in the next cycle a_0 and o_0. This wait is necessary to compute the last p_{size} and p_{size+1} words and the last word o_{size}. The process then begins again for the next word of b, and continues until all words have been computed.

In case of an addition, for the architecture with one multiplier each operand word is added together with the previous carry. Register *RegA* is reset to 1, *RegB* receives one operand and *RegC* receives the other operand. For the architecture with two multipliers the approach is almost the same, except *RegB1* is reset to 1 and it is *RegA1* that receives one of the operands. One downside, is that the computation has to pass through Math Block 2 until the result is written in one of the memories.

Subtraction on the other hand is more difficult for the first architecture. The reason is because it does not only involve the subtraction of both operands, but also an addition with $2n$ [5]. Just as for the addition, *RegA* is reset to 1, *RegB* receives one operand and *RegC* receives the other operand, however the output of the multiplier is negated. After the first word subtraction, instead of computing the next word, the same is fed through the accumulator without the right shift. With the value being fed through the accumulator, both *RegC* and *RegB* receive one word of n, therefore doing the final add of $2n$. The result is kept to be used as carry in the next operation and also stored in memory.

Subtraction in the architecture with two multipliers works similarly as the addition. In the first Math Block, operands words are subtracted and the result goes to the second Math Block. In addition, the second Math Block would add 0, however during subtraction it is used to apply the final addition with $2n$.

In case is necessary to increase the number of extra bits from 4 to a higher value, the value multiplied by n during the subtraction also has to increase. In the architecture with two multipliers, there will be almost no modifications, except one register will receive a different value to reset. In the architecture with one multiplier there will be more changes. The reason is because register $RegA$ has to be reset with value 1 or the other new value, and also $RegC$ will have to be reset to 0.

Table 1. Scheduling for the architecture with one Math Block based on Figure 1 and Algorithm 2

Time	$RegA$	$RegB$	$RegC$	$RegP$
1	b_0			
2	b_0	a_0	0	0
3	b_0	a_1	0	p_0
...		...		...
$l+1$	b_0	a_{l-1}	0	p_{l-2}
$l+2$	n'	0	0	p_{l-1}
$l+3$	n'	0	0	p_l
$l+4$	n'	p_0	0	p_{l+1}
$l+5$	n'	p_0	0	m
$l+6$	m	n_0	p_0	0
$l+7$	m	n_1	p_1	
$l+8$	m	n_2	p_2	o_0
...		...		...
$2l+5$	m	n_{l-1}	p_{l-1}	o_{l-3}
$2l+6$	0	0	p_l	o_{l-2}
$2l+7$	0	a_0	p_{l+1}	o_{l-1}
$2l+8$	b_1	a_0	o_0	o_l
$2l+9$	b_1	a_1	o_1	p_0
...		...		...

The internal memory is configured as a three-port 64x17 (it is 18, but only 17 are necessary) and in little-endian format. For the first architecture, the memory is split into 4 positions of 16 words each, where the first position is reserved for the modulus $n$followed by the constant n'. In case of the second architecture there are two memories and each one can hold up to 2 values of 32 words each. The first memory position is also reserved for constants n and n'. In both cases the other three remaining positions can be used for two inputs and one output variables. The two input variables can share the same address, therefore allowing a square or double operation. However, it is not possible for an input variable to share the address with an output.

During additions and subtractions not all three positions are available as the output. The addition and subtraction operations are implemented as $Out = Out + In$ and $Out = Out - In$. Therefore addition and subtraction operations has to be schedule taken this into account.

In the first architecture, at glance each variable can hold up to 16 words of 17 bits, therefore 272 bits. However, the processor can only work up to 14 words of 17 bits during the multiplication process, therefore 234 bits only, including the

Table 2. Scheduling for the architecture with two Math Blocks based on Figure 2 and Algorithm 3

Time	*RegA1*	*RegB1*	*RegC1*	*RegP1*	*RegM2*	*RegB2*	*RegC2*	*RegP2*
1		b_0						
2	a_0	b_0	0					
3	a_0	b_0	0	p_0				
4	a_0	b_0	0	p_0	p_0	n'	0	
5	a_1	b_0	0	p_0				m
6	a_2	b_0	0	p_1	m	n_0	p_0	0
7	a_3	b_0	0	p_2	m	n_1	p_1	
8	a_4	b_0	0	p_3	m	n_2	p_2	o_0
...		...		...		...		...
$l+3$	a_{l-1}	b_0	0	p_{l-2}	m	n_{l-3}	p_{l-3}	o_{l-5}
$l+4$	0		0	p_{l-1}	m	n_{l-2}	p_{l-2}	o_{l-4}
$l+5$	0	b_1	0	p_l	m	n_{l-1}	p_{l-1}	o_{l-3}
$l+6$	a_0	b_1	o_0	p_{l+1}		0	p_l	o_{l-2}
$l+7$	a_0	b_1	o_0	p_0		0	p_{l+1}	o_{l-1}
$l+8$	a_0	b_1	o_0	p_0	p_0	n'	0	o_l
$l+9$	a_1	b_1	o_1	p_0	p_0	n'	0	m
$l+10$	a_2	b_1	o_2	p_1	m	n_0	p_0	0
$l+11$	a_3	b_1	o_3	p_2	m	n_1	p_1	
$l+12$	a_4	b_2	o_4	p_3	m	n_2	p_2	o_0
...		...		...		...		...

extra 4 bits to avoid reductions during additions [5]. This two words loss in each variable is because the CIOS algorithm needs to store some carry results. For this architecture version, only some ECC curves with less than 234 bits prime fields can work. Within 234 bits it is possible to use prime standard curves of 192 bits or 224 bits [3,14,25]. It is possible to deploy a second memory and increase to 510 bits, including the extra 4 bits, or even more memories.

If the user increases the amount of memory, it would be more interesting to consider the second architecture that can only work up to 527 bits, including the extra 4 bits. In the FIOS algorithm, the intermediate value during multiplication only increases with one word, while in the second architecture increases with two words. Because it can work with 527-bit primes, the second architecture can directly be applied with all standard prime ECC curves published so far. This characteristic makes the second architecture a generic solution, while the first architecture is more suitable for power constrained environments. One possible solution for these extras intermediate values is to store in external registers, thus increasing the number of registers in the entire architecture. However, the prime size for each architecture is not a bottleneck.

Communication between our both architectures is made through 4 instructions and a direct interface with the 64x17 memory. The considered instructions are modular multiplication, addition, subtraction and no operation. During the no operation instruction it is possible to access the 64x17 memory and to read and write at any memory location. To increase the operating frequency, both the input and outputs values are registered.

4 Results and Related Works

The results of both proposed architectures are summed up in Table 3 and 4, together with some literature results. For space and visualization sake, only results for primes which can be directly compared are shown.

From all work shown in Table 3 only our results provide power estimations in Table 4. Those power estimations were obtained by computing $o = (a+b) \cdot (c-d)$ 100 times, for random inputs a, b, c, d in the Microsemi® synthesis tool. The same 100 random inputs and output values generated by a script with SageMath [32] were applied to verify our architecture.

According to Table 4, our architecture's static power does not change, even when the circuit is bigger. This invariable power simulation can be explained by the value precision and the circuit size difference of only 2% of the FPGA's total area. On the other hand, the dynamic power has a higher difference between architectures sizes, but a slight variance between inputs sizes. This small difference can be attributed to a variance in the 100 random samples. For example, for the one multiplier architecture, the average dynamic power is $16.76mW$ and the highest difference is 2% from the average.

With the power and time consumed by each architecture, it is possible to estimate the energy consumed for 224-bit modular multiplication. The first architecture does a multiplication in $2.46\mu s$ with $25.35mW$, therefore it needs $61.35nJ$. The second architecture does the multiplication faster in $1.30\mu s$ with $36.02mW$, thus consuming only $46.83nJ$. The second architecture, when comparing with the first, has an increase of 48% in area, a decrease of 46% in time and an increase of only 42% in terms of power. The power has increased less than the latency, which explains the second architecture being more energy efficient. In some scenarios, like RFID applications, the power consumption is more important than the energy efficiency. In those scenarios architecture 1 is more suitable.

Only a few ECC hardware implementations with prime based finite fields are shown in Table 3. Since some do not provide multiplication results for the same prime size in Xilinx® or Microsemi® FPGAs [2,7,15,27,30,33]. Also, some other works [17,26] have a project strategy have a low latency and/or high throughput approach, so most FPGA resources are required. In contrary, our strategy is to minimize the number of occupied resources, therefore it is not a meaningful comparison.

McIvor et al. [20] analyzed the SOS, CIOS and FIOS algorithms proposed in [18] under the following metrics: area, cycles, latency and throughput for a Virtex II Pro® FPGA. For their analysis the architecture applied was almost the same for all 3 algorithms: an ALU based architecture. In this architecture there is one ALU that can do multiplication or addition, and this ALU keeps iterating over the values. This architecture has an approach that is similar to

Table 3. Comparison of our results to the literature on hardware implementations for ECC. The speed results are for one modular multiplication.

Work	Field	FPGA	LUT4	FF	Emb. Mult.	BRAM 64x18	BRAM 1kx18	Frequency (MHz)	Add. Cycles	Sub. Cycles	Mult. Cycles	Time
Only Finite Field Multiplication												
[20]	128	Virtex II Pro®	1434*	1434*	1	0	0	101.87	–	–	310	1.9
[20]	256	Virtex II Pro®	2866*	2866*	4	0	0	101.87	–	–	582	5.7
[29]	128	Virtex® E	1612*	1612*	0	0	0	97.63	–	–	388†	3.97
[29]	256	Virtex® E	3096*	3096*	0	0	0	100.44	–	–	772†	7.69
[29]	512	Virtex® E	5944*	5944*	0	0	0	95.22	–	–	1540†	16.17
Finite Field Addition, Subtraction and Multiplication												
Our 1	116-132	IGLOO® 2	444	160	1	1	0	200	14	22	184	0.92
Our 1	150-166	IGLOO® 2	444	160	1	1	0	200	16	26	268	1.34
Our 1	184-200	IGLOO® 2	444	160	1	1	0	200	18	30	368	1.84
Our 1	218-234	IGLOO® 2	444	160	1	1	0	200	20	34	484	2.42
Our 2	116-132	IGLOO® 2	658	268	2	2	0	200	16	16	108	0.54
Our 2	150-166	IGLOO® 2	658	268	2	2	0	200	18	18	148	0.74
Our 2	184-200	IGLOO® 2	658	268	2	2	0	200	20	20	200	1.00
Our 2	218-234	IGLOO® 2	658	268	2	2	0	200	22	22	260	1.30
Our 2	252-268	IGLOO® 2	658	268	2	2	0	200	24	24	328	1.64
Our 2	320-336	IGLOO® 2	658	268	2	2	0	200	28	28	488	2.44
Our 2	371-387	IGLOO® 2	658	268	2	2	0	200	31	31	629	3.15
Our 2	405-421	IGLOO® 2	658	268	2	2	0	200	33	33	733	3.67
Our 2	439-455	IGLOO® 2	658	268	2	2	0	200	35	35	845	4.23
Our 2	507-523	IGLOO® 2	658	268	2	2	0	200	39	39	1093	5.47
Finite Field with Inversion												
[10]	128	Virtex II®	3234*	3234*	0	0	0	45.17	1	1	129	2.86
[10]	160	Virtex II®	3708*	3708*	0	0	0	40.28	1	1	161	4.00
[10]	192	Virtex II®	4694*	4694*	0	0	0	35.99	1	1	193	5.36
[10]	224	Virtex II®	5442*	5442*	0	0	0	33.06	1	1	225	6.81
[10]	256	Virtex II®	6218*	6218*	0	0	0	31.92	1	1	257	8.05
Full ECC Processors												
[35]	224	SmartFusion®	3690	3690	0	0	12	109	46	46	360	3.3
[35]	256	SmartFusion®	3690	3690	0	0	12	109	46	46	401	3.7
[35]	224	Virtex II Pro®	1546*	1546*	1	0	3	210	46	46	360	1.7
[35]	256	Virtex II Pro®	1546*	1546*	1	0	3	210	46	46	401	1.9
[35]	224	Virtex II Pro®	2316*	2316*	4	0	3	210	28	28	135	0.64
[35]	256	Virtex II Pro®	2316*	2316*	4	0	3	210	28	28	157	0.75
[36]	256	Virtex II Pro®	3664*	3664*	2	0	9	108.2	44	44	637	5.89

* Maximum possible value assumed from the number of slices. † Values estimated by multiplying time by frequency.

Table 4. Power simulation results for our hardware implementations.

Work	Field	FPGA	LUT4	FF	Emb. Mult.	BRAM 64x18	BRAM 1kx18	Frequency (MHz)	Power (mW) Dyn.	Sta.
Finite Field Addition, Subtraction and Multiplication										
Our 1	116-132	IGLOO® 2	444	160	1	1	0	200	17.09	8.89
Our 1	150-166	IGLOO® 2	444	160	1	1	0	200	16.86	8.89
Our 1	184-200	IGLOO® 2	444	160	1	1	0	200	16.62	8.89
Our 1	218-234	IGLOO® 2	444	160	1	1	0	200	16.46	8.89
Our 2	116-132	IGLOO® 2	658	268	2	2	0	200	26.88	8.89
Our 2	150-166	IGLOO® 2	658	268	2	2	0	200	27.01	8.89
Our 2	184-200	IGLOO® 2	658	268	2	2	0	200	27.11	8.89
Our 2	218-234	IGLOO® 2	658	268	2	2	0	200	27.13	8.89
Our 2	252-268	IGLOO® 2	658	268	2	2	0	200	27.16	8.89
Our 2	320-336	IGLOO® 2	658	268	2	2	0	200	27.08	8.89
Our 2	371-387	IGLOO® 2	658	268	2	2	0	200	27.06	8.89
Our 2	405-421	IGLOO® 2	658	268	2	2	0	200	27.07	8.89
Our 2	439-455	IGLOO® 2	658	268	2	2	0	200	27.00	8.89
Our 2	507-523	IGLOO® 2	658	268	2	2	0	200	26.97	8.89

ours, having one ALU on which all operations are performed. Our strategy differs from theirs in the sense that our architecture is optimized to compute in one cycle $a \cdot b + d + carry$ while theirs performs this operation in more cycles. Also, when they implemented FIOS the same strategy with one ALU was applied, instead of an architecture that benefits more from the algorithm. If we do a direct comparison of implementations, ours is 69% smaller and 52% faster than theirs for 128-bit primes when comparing the architectures with one multiplier. For the architecture with two multipliers for 256-bit primes, ours is 77% smaller and 71% faster than theirs. Part of this improvement can be explained by the FPGA distinct architectures. In the Virtex II Pro® architecture there are only standalone multipliers, while IGLOO®2 has multipliers and adders embedded together, which reduces the amount of area required and increases the operating frequency.

Örs et al. [29] designed a systolic array architecture for Montgomery modular multiplication. It could be said they also applied the FIOS algorithm, since they do multiplication and reduction at the same time. In their approach each cell computes one bit for the entire multiplication process. Their strategy is to compose an array of those cells to compute the modular multiplication, while ours is to iterate on a 17 bits cell. The systolic array is more suitable for platforms, where embedded multipliers are not available directly, like ASICs. It should be noticed that their architecture was devised for the RSA cryptosystem, but it works for ECC as well. By doing a direct comparison, our implementation is 72% smaller and 77% faster for 128-bit primes with one multiplier. For two multipliers and 512-bit primes, our proposal is 89% smaller and 66% faster. This big difference in results is mainly because their architecture doesn't leverage from embedded DSPs, Math Blocks or multipliers.

In the same strategy of not applying any embedded primitive on the FPGA, besides memories, Daly et al. [10] projected a circuit for all prime field operations. The architecture strategy is to make an adder of the entire size of the input, and use the adder iteratively to compute a multiplication. A multiplication can be computed at each iteration by multiplying one bit of one of the inputs by the other entire input. With this strategy an entire multiplier can be made out of one adder. However, in their architecture they opted for two adders for performance reasons. While this strategy reduces greatly the circuit area it also increases operation time, since it is directly proportional with the bit length of the field. This strategy shares some similarities with the systolic array. But in the systolic array some intermediate flip flops can be placed to increase the operating frequency, or it can work in a cyclic way. By comparing directly operations on 224-bit primes and different FPGAs, our proposal is 92% smaller and 52% faster for one multiplier. Their proposal has the inversion operation, while our proposal features all necessary operations to implement.

In more recent work Varchola et al. [35] proposed a small ECC architecture for only NIST primes curves. The work has the same objective to make an ECC architecture as small as possible, therefore using only a small number of DSPs, if available, were demanded. Their architecture follows the same approach as ours, of one ALU that keeps receiving values to be processed. Their ALU is also one multiplier, using the built-in FPGA multiplier if it is available, and followed by final addition that is done through FPGA LUTs. In their case two FPGA implementations were made, one with a SmartFusion® FPGA from Microsemi® and another one for a Xilinx® Virtex II Pro®. Since SmartFusion does not have embedded multipliers, none were deployed, while they were exploited on Virtex II Pro®. Since the circuit area is for the entire ECC co-processor with scalar point multiplication, only the timing results for modular multiplication can be compared. For the SmartFusion our architecture is slightly better, but for the Virtex II Pro® our proposal is slower. When comparing the latency, their proposal is 30% faster than ours with one multiplier for 224 bits and 54% faster than ours for two multipliers. The reason for this difference is a more efficient modular reduction algorithm, that only works for Solinas primes.

Another recent work with the same minimization strategy was conducted by Vliegen et al. [36]. The strategy applied was also to reduce the number of memories and embedded multipliers on a Virtex II Pro® FPGA. For their architecture the CIOS algorithm was followed in a straightforward manner. For this reason, two multipliers were used, instead of 1 like in our proposal. More than one circuit was made, there were 3 circuits of the same architecture, but each one with two bigger basic multipliers. A comparison of the cycle count of our two-multiplier architecture following the FIOS algorithm and their architecture for 256 bits, shows a speed-up of 49% in favor of our solution. This speed-up is achieved thanks to a more efficient scheduling of the operations on the two multipliers.

5 Final Considerations

We have shown two different architectures with the same approach of minimizing the number of multipliers and memories. While our first proposal is smaller and more power efficient, our second proposal is faster and can work with primes up to 523 bits. Both architectures have advantages and disadvantages, and therefore can be used in different scenarios.

A comparison of our work with the literature shows a considerable reduction of both the resources and the latency for a Montgomery multiplication allowing generic primes. Moreover, we managed to achieve these results on an FPGA based on a low power Flash technology, namely the Microsemi IGLOO 2 FPGA. This makes our 1-multiplier and 2-multiplier solutions suitable for power and energy efficient applications, respectively. The most important contribution of our work is the optimal scheduling of the CIOS and FIOS algorithms on the internal architecture of the IGLOO 2. This results in a minimization of the number of occupied LUTs and Flip Flops in combination with a minimization of the number of cycles consumed for one multiplication.

For future work, our proposal is to integrate this solution into a bigger co-processor that can do ECC scalar multiplication. The main challenge will be to minimize the additional memory and control resources needed to store and process the intermediate values in the computation.

References

1. Agence nationale de la sécurité des systèmes d'information: Mécanismes cryptographiques - Règles et recommandations. Tech. rep., ANSSI (2014)
2. Alrimeih, H., Rakhmatov, D.: Fast and Flexible Hardware Support for ECC Over Multiple Standard Prime Fields. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 22(12), 2661–2674 (Dec 2014)
3. Aranha, D.F., Barreto, P.S.L.M., Pereira, G.C.C.F., Ricardini, J.: A note on high-security general-purpose elliptic curves (2013), http://eprint.iacr.org/2013/647
4. Barrett, P.: Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In: Advances in Cryptology - CRYPTO' 86, Lecture Notes in Computer Science, vol. 263, pp. 311–323. Springer Berlin Heidelberg (1987)
5. Batina, L., Muurling, G.: Montgomery in Practice: How to Do It More Efficiently in Hardware. In: Topics in Cryptology — CT-RSA 2002, Lecture Notes in Computer Science, vol. 2271, pp. 40–52. Springer Berlin Heidelberg (2002)
6. Bernstein, D.J.: Curve25519: new Diffie-Hellman speed records. In: Public key cryptography—PKC 2006, 9th international conference on theory and practice in public-key cryptography. Springer, New York, NY, USA (2006)
7. Blum, T., Paar, C.: High-radix Montgomery modular exponentiation on reconfigurable hardware. IEEE Transactions on Computers 50(7), 759–764 (Jul 2001)
8. Chen, G., Bai, G., Chen, H.: A High-Performance Elliptic Curve Cryptographic Processor for General Curves Over GF(p) Based on a Systolic Arithmetic Unit. Circuits and Systems II: Express Briefs, IEEE Transactions on 54(5), 412–416 (May 2007)
9. Crandall, R.E.: Method and apparatus for public key exchange in a cryptographic system (1992), u.S. Patent number 5159632

10. Daly, A., Marnane, W., Kerins, T., Popovici, E.: An FPGA implementation of a GF(p) ALU for encryption processors. Microprocessors and Microsystems 28(5–6), 253 – 260 (2004), special Issue on FPGAs: Applications and Designs
11. Diffie, W., Hellman, M.E.: New directions in cryptography. Information Theory, IEEE Transactions on 22(6), 644–654 (1976)
12. nationale de la sécurité des systèmes d'information, A.: Publication d'un paramétrage de courbe elliptique visant des applications de passeport électronique et de l'administration électronique française. Tech. rep., ANSSI (2011)
13. Eberle, H., Gura, N., Shantz, S., Gupta, V., Rarick, L., Sundaram, S.: A public-key cryptographic processor for RSA and ECC. In: Application-Specific Systems, Architectures and Processors, 2004. Proceedings. 15th IEEE International Conference on. pp. 98–110 (Sept 2004)
14. ECC Brainpool: ECC Brainpool standard curves and curve generation. Tech. rep., Brainpool (2005)
15. Ghosh, S., Alam, M., Chowdhury, D.R., Gupta, I.S.: Parallel crypto-devices for GF(p) elliptic curve multiplication resistant against side channel attacks. Computers & Electrical Engineering 35(2), 329 – 338 (2009), circuits and Systems for Real-Time Security and Copyright Protection of Multimedia
16. J. Jonsson and B. Kaliski: Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. Tech. rep., RSA Laboratories (2003)
17. Javeed, K., Wang, X.: Efficient Montgomery Multiplier for Pairing and Elliptic Curve Based Cryptography. In: Communication Systems, Networks Digital Signal Processing (CSNDSP), 2014 9th International Symposium on. pp. 255–260 (July 2014)
18. Koç, Ç.K., Acar, T., Kaliski Jr., B.S.: Analyzing and comparing Montgomery multiplication algorithms. Micro, IEEE 16(3), 26–33 (Jun 1996)
19. Kocher, P., Jaffe, J., Jun, B.: Differential Power Analysis. In: Advances in Cryptology – CRYPTO' 99, Lecture Notes in Computer Science, vol. 1666, pp. 388–397. Springer Berlin Heidelberg (1999)
20. McIvor, C., McLoone, M., McCanny, J.V.: FPGA Montgomery multiplier architectures - a comparison. In: Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on. pp. 279–282 (April 2004)
21. Microsemi: IGLOO2 Product Information Brochure. Tech. rep., Microsemi (2014), `http://www.microsemi.com/document-portal/doc_download/132013-igloo2-product-information-brochure`
22. Miller, V.: Use of Elliptic Curves in Cryptography. In: Advances in Cryptology - CRYPTO 85 Proceedings, Lecture Notes in Computer Science, vol. 218, pp. 417–426. Springer Berlin / Heidelberg, Berlin, Germany (1986)
23. Montgomery, P.L.: Modular Multiplication without Trial Division. Mathematics of Computation 44(170), 519–521 (1985)
24. National Institute for Standards and Technology: Federal information processing standards publication 186-2. digital signature standard. Tech. rep., NIST (2000)
25. National Institute for Standards and Technology: Federal information processing standards publication 186-4. digital signature standard. Tech. rep., NIST (2013)
26. Orlando, G., Paar, C.: A Scalable GF(p) Elliptic Curve Processor Architecture for Programmable Hardware. In: Cryptographic Hardware and Embedded Systems — CHES 2001, Lecture Notes in Computer Science, vol. 2162, pp. 348–363. Springer Berlin Heidelberg (2001)
27. Pöpper, C., Mischke, O., Güneysu, T.: MicroACP - A Fast and Secure Reconfigurable Asymmetric Crypto-Processor. In: Reconfigurable Computing: Architec-

tures, Tools, and Applications, Lecture Notes in Computer Science, vol. 8405, pp. 240–247. Springer International Publishing (2014)

28. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 21(2), 120–126 (feb 1978)
29. Örs, S.B., Batina, L., Preneel, B., Vandewalle, J.: Hardware implementation of a Montgomery modular multiplier in a systolic array. In: Parallel and Distributed Processing Symposium, 2003. Proceedings. International. p. 8 (April 2003)
30. Sakiyama, K., Mentens, N., Batina, L., Preneel, B., Verbauwhede, I.: Reconfigurable Modular Arithmetic Logic Unit for High-Performance Public-Key Cryptosystems. In: Reconfigurable Computing: Architectures and Applications, Lecture Notes in Computer Science, vol. 3985, pp. 347–357. Springer Berlin Heidelberg (2006)
31. Solinas, J.A.: Generalized Mersenne Numbers. Tech. rep., Center for Applied Cryptographic Research, University of Waterloo (1999)
32. Stein, W., et al.: Sage Mathematics Software (Version 7.0). The Sage Development Team (2016), `http://www.sagemath.org`
33. Tamura, S., Yamada, C., Ichikawa, S.: Implementation and Evaluation of Modular Multiplication Based on Coarsely Integrated Operand Scanning. In: Proceedings of the 2012 Third International Conference on Networking and Computing. pp. 334–335. ICNC '12, IEEE Computer Society, Washington, DC, USA (2012)
34. Tenca, A.F., Koç, Ç.K.: A Scalable Architecture for Montgomery Multiplication. In: Cryptographic Hardware and Embedded Systems, Lecture Notes in Computer Science, vol. 1717, pp. 94–108. Springer Berlin Heidelberg (1999)
35. Varchola, M., Güneysu, T., Mischke, O.: MicroECC: A Lightweight Reconfigurable Elliptic Curve Crypto-processor. In: Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on. pp. 204–210 (Nov 2011)
36. Vliegen, J., Mentens, N., Genoe, J., Braeken, A., Kubera, S., Touhafi, A., Verbauwhede, I.: A compact FPGA-based architecture for elliptic curve cryptography over prime fields. In: Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on. pp. 313–316 (July 2010)
37. Walter, C.: Montgomery exponentiation needs no final subtractions. Electronics Letters 35(21), 1831–1832 (Oct 1999)