A compact FPGA-based microcoded coprocessor for exponentiation in asymmetric encryption

L. Rodriguez-Flores*, Miguel Morales-Sandoval†, R. Cumplido*, C. Feregrino-Uribe*, I. Algredo-Badillo‡

*Instituto Nacional de Astrofisica, Optica y Electronica (INAOE), Tonantzintla, Mexico
lrodriguez@inaoep.mx
†Cinvestav Unidad Tamaulipas, Mexico
‡Ingenieria en Tecnologias de la Informacion, Universidad Politecnica de Tlaxcala, Mexico

***Abstract*—Exponentiation in multiplicative groups is the most time-consuming and critical operation for implementing asymmetric cryptography for key exchange, digital signatures, and digital envelopes in security protocols. Most of the designs previously reported to support this operation are mainly devoted to achieve the highest performance. However, in current computing paradigms highly dominated by interconnected small, computing-constrained devices, small but efficient cryptographic hardware designs are better preferred. This paper presents a novel coprocessor for exponentiation, having as main design goal to have a compact design well suited for constrained computing environments. As key aspect, the proposed coprocessor uses a microprogramming approach together with a pipelined datapath that processes operands digit-by-digit. The coprocessor is designed to exploit the available resources in modern FPGAs, thus reducing the usage of standard logic. The experimental results reveal that our design achieves better efficiency than other implementation approaches.**

I. Introduction

Asymmetric (public key) cryptosystems, such as RSA, DSA, DH, and ElGammal, base their security in the logarithm discrete problem [1]. For a multiplicative group $\mathbb{F}_p^\star = \{1, 2, ...p-1\}$ with p a prime number, the discrete logarithm problem is defined as: Given a generator g and an element y in $\mathbb{F}_p^\star$, find the smallest positive integer x such that $g^x = y \bmod p$. This problem has been considered computationally hard [1]. As $\mathbb{F}_p^\star$ is a subset of the finite field $\mathbb{F}_p$, it is common to refer to exponentiation in $\mathbb{F}_p^\star$ or $\mathbb{F}_p$ indistinctly.

The exponentiation g^x is the most time-consuming operation in public key cryptosystems. That is why several custom hardware modules have been proposed to accelerate this costly operation, most of the related works have had the main design goal of high performance and throughput [2].

However, with the advent of new computing paradigms as the Internet of Things (IoT), recently a design goal in cryptographic hardware is to create hardware modules with small area resources and more efficient designs, with better usage of the constrained computing resources available.

Field Programmable Gateway Arrays (FPGAs) are already used in emerging IoT applications such as Smart City infrastructure, Smart Grid, intelligent factory automation, etc [3]. Furthermore, FPGAs have been successful implementation platforms for cryptography algorithms, and for compact designs [4]. Compact designs in modern FPGAs is mainly achieved by taking advantage of the high-performance capabilities of embedded IP-cores such as DSP48E and Block Ram memories (BRams) [5], [6].

In this paper, we explore the design of compact hardware for $\mathbb{F}_p$ exponentiation having as key features:

- The implementation of a low area datapath that processes the digits of the operands digit-by-digit. This allows to propose a hardware architecture with a simplified datapath. When this design is implemented in the modern FPGAs recommended for IoT applications, less standard logic is used but the embedded IP blocks are used as much as possible.
- A more compact design is obtained by implementing the control unit using a microprogramming approach. $\mathbb{F}_p$ exponentiation hardware found in the literature usually implement the control logic as a FSM to orchestrate the dataflow and algorithm execution. However, in this work we demonstrate that the microprogramming approach is a good alternative since the control unit is simplified and the associated area reduced.

The proposed design was described in VHDL, prototyped and validated for the Xilinx Virtex 7 FPGA. Different operand and digit sizes were considered during experimentation, leading to various configurations for the memory blocks and datapath. As a result, it was possible to find those configurations that achieve smaller area, higher throughput, or higher efficiency.

The rest of this paper is organized as follows: Section II briefly reviews exponentiation in $\mathbb{F}_p$. The proposed compact architecture for $\mathbb{F}_p$ exponentiation is presented in Section III. Section IV describes and discusses the experimentation results. Finally, Section V summarizes contributions of this work and gives directions for future work.

II. $\mathbb{F}_p$ exponentiation

Modular exponentiation is often implemented using the Montgomery Powering Ladder exponentiation (MPL) algorithm (Algorith 1) that is a constant time algorithm resistant to Simple Power Analysis attacks [7].

The modular multiplication is the underlying operation in $\mathbb{F}_p$ exponentiation. The Montgomery method [8] has been used extensively for modular multiplication because it uses simpler

978-1-5090-5859-4/17/$31.00 ©2017 IEEE

and less costly operations as additions and shifts, well suited for hardware implementations.

A. Montgomery method

Montgomery multiplication is defined as follows. Let the modulus p be an integer number of N-bits, $2^{N-1} \leq p < 2^N$. Let R and p being relative primes, this means $\gcd(p, R) = 1$. Thus, it is possible to compute the numbers R^{-1} and p' using the identity $R \times R^{-1} + p \times p' = 1$, with $0 < R^{-1} < p$ and $0 < p' < R$, using methods such as the extended Euclidean Algorithm.

The Montgomery product of two numbers $A, B \in \mathbb{F}_p$ is defined by equation 1, where A', B' are the Montgomery numbers corresponding to A, B respectively, and $A' = A \times R \bmod p$, $B' = B \times R \bmod p$.

$$\text{MMult}(A, B) = C' = A' \times B' \times R^{-1} \mod p \qquad (1)$$

The Montgomery algorithm has a significant advantage when many consecutive multiplications are required, since number conversions will be only required at the beginning and the end of the accumulative multiplications, such as in $\mathbb{F}_p$ exponentiation.

Algorithm 1 Montgomery Powering Ladder algorithm for $\mathbb{F}_p$ exponentiation

```
Require: $m \in \mathbb{F}_p$
Require: $e = (e_{L-1}, \cdots, e_0) \in \mathbb{N}$
Ensure: $m^e \mod p$
 1: $R0 \leftarrow 1$;
 2: $R1 \leftarrow m$;
 3: for $i = L - 1$ downto 0 do
 4:     if $e_i == 1$ then
 5:         $R0 \leftarrow R0 \times R1 \mod p$;
 6:         $R1 \leftarrow R1 \times R1 \mod p$;
 7:     else
 8:         $R0 \leftarrow R0 \times R0 \mod p$;
 9:         $R1 \leftarrow R1 \times R0 \mod p$;
10:     end if
11: end for
12: return $R0$;
```

B. Digit-digit $\mathbb{F}_p$ exponentiation

Due to the large operands size (1024-4096 bits) usually required in some logarithm discrete based cryptosystems, parallel implementations of $\mathbb{F}_p$ exponentiation consume a high number of area resources and thus of energy. One approach to shrink down the hardware designs is not to process operands in parallel but iteratively by means of a digit-based processing approach [9], [6].

A small, digit-based design for $\mathbb{F}_p$ multiplication, as the one proposed in [9] can be used to compute the partial multiplications in lines 5,6 and 8,9 of Algorithm 1. The design in [9] uses a digit-by-digit processing approach, which has the following advantages:

- Operands can be stored in memory blocks, each operand's digit stored in a memory word (sequential data access but parallel processing).
- The datapath complexity will depend on the digit size instead of the operand size (scalable design).
- Smaller width of datapath and processing modules i.e., multipliers (compact designs).

III. Hardware design for digit-digit $\mathbb{F}_p$ exponentiation

In this work, we retake the digit-by-digit processing approach proposed in [9] and apply it to construct a functional $\mathbb{F}_p$ exponentiator, well suited to accelerate that costly operation in cryptographic protocols.

The drawback with the algorithm proposed in [9] is the use of a shift register to store the partial results of the multiplication. Since the result (in a shift register) must be used as the next input operand (in block memory), it would be used more logic than needed and additional latency will be required to transfer data from the shift register to the memory at each iteration.

In order to serves as a building block for constructing the $\mathbb{F}_p$ exponentiatior, in this work we redesign the compact multiplier presented in [9]. Main changes include:

- Removing the shift register and use a memory block to store the digits of partial results.
- Modification of the control logic in the multiplier, allowing that the memory block for the partial multiplication reads and writes in the same memory address.
- The frequency of the hardware implementation was improved by adding more register for specific variables in the digit-by-digit processing algorithm.

Algorithm 2 is the modified version of the algorithm originally presented in [9]. From this algorithm, we provide a new functional hardware architecture (Figure 1) for Montgomery multiplication that keeps all the operand and result in external memory blocks and perform the processing under a digit-by-digit fashion. The result is obtained iteratively and stored in memory A.

Algorithm 2 Digit-digit Montgomery algorithm

```
Require: $X = \sum_{i=0}^{n-1} X_i\beta^i$, $Y = \sum_{i=0}^{n-1} Y_i\beta^i$, $p = \sum_{i=0}^{n-1} p_i\beta^i$,
    $0 < X, Y < 2 \times p$, $R = \beta^n$ with $p' = -p^{-1} \mod \beta$
Ensure: $A = \sum_{i=0}^{n-1} a_i\beta^i = X \times Y \times R^{-1} \mod p$
 1: $A \leftarrow 0$;
 2: for $i \leftarrow 0$ to $n - 1$ do
 3:     $c_0 \leftarrow 0$
 4:     for $j \leftarrow 0$ to $n - 1$ do
 5:         $s_j \leftarrow [A_j + X_j \times Y_i]$
 6:         if $j = 0$ then
 7:             $q_i \leftarrow (s_j \times p') \mod \beta$
 8:         end if
 9:         $r_j \leftarrow q_i \times p_j$
10:         $\{c_{j+1}, t_j\} \leftarrow s_j + r_j + c_j$
11:         if $j > 0$ then
12:             $A_{j-1} \leftarrow t_j$
13:         end if
14:     end for
15:     $A_{n-1} \leftarrow c_n$
16: end for
17: return $A$;
```

Although the digit-by-digit processing approach simplifies the design, the control logic for data routing and algorithm execution becomes more elaborated. The control block is required to manage memory addresses, multiplexers, set/reset registers, and enable/disable write memories.

The MPL architecture can be implemented using two independent Montgomery datapath multipliers. At each iteration, it is required to compute one square and one multiplication

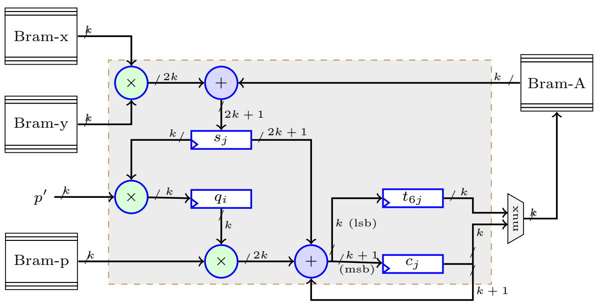

Fig. 1. Datapath for the Montgomery hardware implementation.

TABLE I
MICROCODE FOR THE CONTROL MODULE.

addr-x	addr-p	addrA	wrA	rst-cj	load-qi	mux-cj	to store
0	3f	3b	1	0	0	0	0x0ffb8
0	0	3c	1	0	0	0	0x003c8
1	0	3d	1	0	0	0	0x103d8
2	1	3e	1	1	1	0	0x207ee
3	2	3f	1	1	0	1	0x30bfd
4	3	3f	0	0	0	0	0x40ff0
5	4	0	1	0	0	0	0x51008
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
3e	3d	39	1	0	0	0	0x3ef798
3f	3e	3a	1	0	0	0	0x3ffba8

over different parameters. The first multiplier always takes $R0$ as one input, and the second multiplier always takes $R1$. The second input operand in each multiplier depends on the current bit tested in the exponent. Additionally, to implement the square operation, dual port memories are required because the same operand in memory must be assigned to the two input buses in the multiplier.

Furthermore, it is not possible to overwrite the memories of operands, operands must be kept and used in the next iterations of Algorithm 2, at least during an entire i iteration. Consequently, two temporary memories $R00$ and $R11$ are required. Firstly, the operands are stored in $R0$ and $R1$ and the partial results are written in $R00$ and $R11$, but in the next iteration, $R00$ and $R11$ now store the operands and $R0$ and $R1$ the partial results.

At each partial Montgomery multiplication control signals and memory addresses are the same for both datapaths, thus a single control logic module can be used and shared by both Montgomery multipliers. Another control logic module is still needed to orchestrate the dataflow of Algorithm 1 and control signals of the four memories (R0, R1, R00, R11). The MPL architecture design is shown in Figure 2.

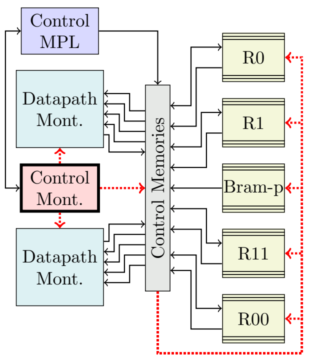

Fig. 2. Digit-digit compact MPL architecture.

A. Control block

In order to analyze the impact of the control logic module in the entire hardware architecture for the $\mathbb{F}_p$ exponentiator, two implementation options were followed: the first one is the classic used based on finite state machine (FSM) and the second one was the microcode approach.

TABLE II
IMPLEMENTATION RESULTS FOR THE DATAPATH BLOCK (1024 BITS OPERAND SIZE)

Digit	Slices	BRams	DSP	Freq. (MHz)
16	16	0	4	232.61
32	66	0	11	135.64
64	230	0	33	97.66

The control module for the Montgomery multiplier implemented with a FSM only depends on the actual state. The FSM is mainly constructed with counters, comparators, and registers. An FSM implementation requires a considerable amount of standard logic to implement all these components together with the logic to enable or disable control signals in those components depending on the current state. Contrary, a microprogramming approach stores the necessary signals in a memory, and only logic to read these instructions is needed as well as a memory block to store the microprogram.

Suppose that the operand has $size = 1024$ and the digit size is $k = 16$, so there are $1024/16 = 64$ digits per operands, all of them sequentially accessed but processed in parallel. Since the latency in the Montgomery multiplier is $64{*}65{+}4 = 4162$, a Montgomery multiplication, for this example, takes 4162 cycles. Note that at each i iteration of Algorithm 2, the control signals are identical. Consequently, it is possible to store only the signals for one iteration and read them many times. In the previous example, it is required to store only 65 control signals and read them 64 times. The microcode for the architecture in Figure 2 is presented in the Table I. The address of the Bram-y memory is used as the index to count how many times the microcode is read. We propose to use the block memory that stores the operand P to store the microcode, and use a dual port memory; one port to read the operand P and the other to read the microcode.

IV. Implementation Results

All the design of the MPL was modeled in VHDL, validated in simulations with Modelsim 10.4, and synthesized for the Virtex 7 Xilinx FPGA using ISE 14.7. The datapath and the control logic (FSM and microprogramming) modules of the exponentiator were independently implemented obtaining the results shown in Tables II and III, respectively.

TABLE III
IMPLEMENTATION RESULTS FOR THE CONTROL BLOCK, USING THE MICROPROGRAMMING AND FSM APPROACHES (1024 BITS OPERAND SIZE)

Digit	Slices	BRams	DSP	Freq. (MHz)
16	8	0	0	447.42
32	9	0	0	490.67
64	9	0	0	397.93
16 (FSM)	34	0	0	227.58
32 (FSM)	20	0	0	339.21
64 (FSM)	15	0	0	398.40

Most of the logic in the datapath are multipliers and adders which are implemented by the embedded DSP modules. Contrary to the datapath, the hardware resources for the control unit decrease with bigger digits in the FSM approach, but in the microprogramming approach the resources do not vary.

The MPL exponentiator architecture was implemented with $k = 16, 32, 64$ bits digit sizes and 1024 bits operand size, obtaining the results shown in Table IV. The microprogramming control reduces the required slices and increments the efficiency. With $k = 16$ is obtained the best efficiency. Thus, 2048 and 4096 bits operand sizes were tested with $k = 16$ digit size, shown in Table V. From these results, it seems that the digit size affects the required slices more than the operand size. If the operand size is incremented, he datapath does not need to be modified, but the control logic needs to use bigger counters and comparators. Invariably, the architecture requires more clock cycles when the operands' size increases.

TABLE IV
IMPLEMENTATION RESULTS FOR MPL ARCHITECTURE, USING THE MICROPROGRAMMING AND FSM APPROACHES (1024 BITS OPERAND SIZE)

Digit	Slices	BRams	DSP	Freq. (MHz)	Throughput (Mbps)	Efficiency (Kbps/slice)
16	84	6	8	208.33	0.050	0.595
32	213	6	22	109.20	0.102	0.482
64	607	11	66	89.62	0.322	0.531
16 (FSM)	104	6	8	192.41	0.046	0.444
32 (FSM)	228	6	22	99.60	0.093	0.411
64 (FSM)	574	10	66	80.21	0.288	0.503

TABLE V
IMPLEMENTATION RESULTS FOR THE MPL WITH MICROPROGRAMMING APPROACH (16 BITS DIGIT SIZE)

Size	Slices	BRams	DSP	Freq. (MHz)	Throughput (Mbps)	Efficiency (Kbps/slice)
1024	84	6	8	208.33	0.050	0.595
2048	105	6	8	207.38	0.025	0.239
4096	95	6	8	210.08	0.012	0.134

Table VI show a comparison with the state of the art works. Okzusoglu [5] presents a compact hardware architecture for a modular exponentiation based on the Montgomery method and MPL, using a systolic array approach. To the author's knowledge, [6] is the most compact implementation of $\mathbb{F}_p$ exponentiation. However, our proposed architecture is smaller than [6] and outperforms implementations in [5] [6] in terms of efficiency.

TABLE VI
RESULTS AND COMPARISON FOR A 1024-BIT EXPONENTIATION.

Work	Alg.	FPGA	Area (slices)	Freq. (MHz)	Thrg (Kbps)	Efficiency (Kbps/slice)
[5]	MPL	Spartan3E	3899	119.05	128.84	0.03
[2](k=2)	LSB	Virtex-5	4060	384.62	503.60	0.12
[10]	MPL	Virtex-5	3218	346.02	322.01	0.10
[11]	LSB	Virtex-5	6776	401.00	747.40	0.11
[6](k=32)	LSB	Virtex-7	1060	485.00	439.48	0.41
our (k=16)	MPL	Virtex-7	**84**	208.33	50.00	**0.59**
our (k=32)	MPL	Virtex-7	213	109.20	102.00	0.48

V. CONCLUSIONS

A compact design for exponentiation in $\mathbb{F}_p$, well suited for implementing asymmetric encryption over constrained small computing devices was discussed. A key feature to achieve compactness without loss of efficiency was to use a digit-by-digit computing approach and implement the control logic using microcoding. The proposed hardware design can be used in embedded security applications based on the discrete logarithm problem. Further research can be conducted to use the ideas presented here to obtain a compact design for cryptosystems based on the Elliptic Curve Discrete Logarithm Problem.

REFERENCES

[1] K. S. McCurley, "The discrete logarithm problem," in *Cryptology and Computational Number Theory*, ser. Proceedings of Symposia in Applied Mathematics, C. Pomerance, Ed., vol. 42. Providence: American Mathematical Society, 1990, pp. 49–74.

[2] G. Sutter, J. Deschamps, and J. Imana, "Modular multiplication and exponentiation architectures for fast RSA cryptosystem based on digit serial computation," *IEEE Transactions on Industrial Electronics*, vol. 58, no. 7, pp. 3101–3109, July 2011.

[3] K. Morris, "Fpgas in the IoT," *Electronic Engineering Journal*, 2014.

[4] D. B. Roy, P. Das, and D. Mukhopadhyay, *ECC on Your Fingertips: A Single Instruction Approach for Lightweight ECC Design in GF(p)*. Cham: Springer International Publishing, 2016, pp. 161–177.

[5] E. Oksuzoglu and E. Savas, "Parametric, secure and compact implementation of RSA on FPGA," in *International Conference on Reconfigurable Computing and FPGAs*, 2008, pp. 391–396.

[6] I. San and N. At, "Improving the computational efficiency of modular operations for embedded systems," *Journal of Systems Architecture*, vol. 60, no. 5, pp. 440 – 451, 2014.

[7] P. C. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '99. London, UK, UK: Springer-Verlag, 1999, pp. 388–397.

[8] P. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[9] M. Morales-Sandoval and 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*, ser. GLSVLSI '13. New York, NY, USA: ACM, 2013, pp. 245–250.

[10] T. Wu, S. Li, and L. Liu, "Fast, compact and symmetric modular exponentiation architecture by common-multiplicand Montgomery modular multiplications," *Integration, the VLSI Journal*, vol. 46, no. 4, pp. 323 – 332, 2013.

[11] A. Rezai and P. Keshavarzi, "High-throughput modular multiplication and exponentiation algorithms using multibit-scan-multibit-shift technique," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 9, pp. 1710–1719, Sept 2015.