Compact and Flexible Microcoded Elliptic Curve Processor for Reconfigurable Devices

Samuel Antão, Ricardo Chaves, and Leonel Sousa
Instituto Superior Técnico/INESC-ID
http://sips.inesc-id.pt/
{sfan,rjfc,las}@sips.inesc-id.pt

Abstract

This paper presents a very compact and flexible processor to support Elliptic Curve (EC) cryptosystems based on $GF(2^m)$ finite fields. This processor can be customized with a two level micro-instructions hierarchy that allows the customization of both field operations and EC algorithms. It was specially designed to benefit from reconfiguration capabilities to scale arithmetic units for different sizes and to replicate processing units to enhance performance. The flexibility resulting from these characteristics was not found in the related art. The proposed processor was implemented and thoroughly tested in a Xilinx Virtex XC4VSX35 supporting a real EC algorithm for point multiplication for a field $GF(2^{163})$, requiring $1.35ms$ and at most 15 times less area than related implementations.

1. Introduction

EC cryptography was proposed in 1985 by N. Koblitz and V. Miller [5, 9]. Since then, it has been gathering increasing importance. Although other algorithms, such as RSA, have reached a higher popularity and are widely used, a new set of circumstances motivates the need to research a different cryptosystem. Among these new circumstances are the increasing use of portable devices, with constrained computing and memory, and with the increasingly more valuable bandwidth and power resources. EC cryptosystems appear as an interesting alternative, since its arithmetic is computationally more efficient and presents higher security per key bit [6]. Several implementations of EC systems have been proposed for both $GF(p)$ and $GF(2^m)$ finite fields, with p a prime and m an integer. $GF(2^m)$ has been leading to more efficient hardware solutions, due to its mathematical properties [1].

In this paper, a flexible and compact processor architecture is proposed to implement $GF(2^m)$ arithmetics in order to support operations over non-supersingular EC. This processor is based in microcoded instructions for basic $GF(2^m)$ operations, used to program custom algorithms for different applications. We introduced micro-instructions with different complexity levels, not only for computing the $GF(2^m)$ field level operations but also to customize these operations supported in the computation over partitions (words) of a field element binary representation. Another characteristic of the proposed processor is its compactness, allowing it to be implemented in area constrained devices. This processor was also specially adapted to reconfigurable devices, namely FPGAs. Exploring the reconfigurable capabilities of these devices allows to scale the arithmetic units to different field sizes (different security levels) and the replication of processing units in order to obtain higher throughputs. Moreover, this can be done dynamically in the nowadays FPGA technologies.

The flexibility of the proposed processor achieved with the word and field microcode, and from the modularity that allows for an easy replication, is not present in the related art. The related art implementations supported in micro-instructions only provide adaptability for particular operations, namely field multiplication, where parallelism [6] or the pipelining levels [3] can be customized. The proposed processor is autonomous in the sense that while computing the programed algorithms no exterior information is required, which is not the case of the known existing related art [4].

In this paper, the proposed processor is used to implement EC point multiplication based on a fast Karatsuba-Offman $GF(2^m)$ multiplication algorithm. The implemented EC point multiplication algorithm has the particularity of computing the result represented with the size of only one of the two coordinates of an EC point. This compact representation allows us to save half the bandwidth comparing with the traditional implementations, which require both the EC point coordinates [1]. The evaluation of this algorithm in the proposed processor suggests that, with at most half the area of the existing relating art, high throughputs are still achieved.

The paper is organized as follows. Section 2 presents an overview of the EC and field arithmetics. The implementation details of the proposed processor are presented in Section 3. Section 4 describes the algorithms used to test the proposed processor and in Section 5 results are presented. Finally, section 6 give some concluding remarks and directions for future work.

2. Elliptic Curve Arithmetic

The arithmetic presented in this paper is dedicated for non-supersingular elliptic curves. A non-supersingular EC is a set composed by a point at infinity $\mathcal{O}$ and the points $P_i = (x_i, y_i) \in GF(2^m) \times GF(2^m)$ that respects the following equation:

$$y_i^2 + x_i y_i = x_i^3 + a x_i^2 + b,\ a, b \in GF(2^m). \quad (1)$$

A commutative addition operation can be established among the points in an EC, composing a commutative group $E(m, a, b)$. The exponentiation of an EC point over this group is obtained by applying recursively the group operation (point addition) as:

$$\underbrace{Q_i = P_i + P_i + \ldots + P_i}_{k \text{ times}}. \quad (2)$$

(2) can also be represented as a point multiplication by a scalar $Q_i = kP_i$. Computing k knowing Q_i and P_i is known to be computational hard (the complexity increases exponentially with the size of k). The security of this cryptosystem is supported on this operation, which is called Elliptic Curve Discrete Logarithm Problem (ECDLP). Since the EC described in (1) is supported over $GF(2^m)$, the EC addition operation consists of several field operations.

2.1 Field Arithmetic

As mentioned, in this work the $GF(2^m)$ finite field is used. Two operations are defined over this field: addition and multiplication. The inverse of these operations are also defined. In order to distinguish between the EC operations and field operations, the EC addition and multiplication are onwards designated as point addition and point multiplication, while the operations over $GF(2^m)$ are designated as field addition and multiplication.

The field $GF(2^m)$ is composed by a binary field $GF(2)$ extended in m dimensions, with its elements represented by m bit vectors. Therefore, the field addition is defined as a set of bitwise XOR operations, which allows the $GF(2^m)$ arithmetic to be efficiently implemented in hardware.

The field $GF(2^m)$ is usually represented with a basis where each bit of the binary representation of a field element index a different element of the basis. The most used basis are normal (or optimal normal) and polynomial basis. The normal basis is composed by a set of vectors $(\alpha^{2^0}, \alpha^{2^1}, \ldots, \alpha^{2^{m-1}})$, where α is a finite field element. Using this basis, operations such as squaring are computed with a left rotational shift operation, which can be very efficiently implemented. For some values of m an optimal normal basis exists and a Massey-Omura multiplier can be constructed, where the dependencies of each output bit from the input operands are minimal [7].

Another type of basis is the polynomial basis, which consist of m powers of a m order irreducible polynomial root (x): $x^0, x^1, \ldots, x^{m-1}$. The reduction modulo the irreducible polynomial of an operation result to a m bit result can be efficiently performed using this basis. This allows different approaches to implement algorithms over this basis. For example, there are multiplication algorithms that embed the reduction modulo of the irreducible polynomial, such as in the Mastrovito multiplier [3], and there are others for which the Karatsuba-Offman multiplication algorithm is computed and an individual reduction step is applied at the end [11]. This means that this basis allows for more flexible implementations. Furthermore, a polynomial basis exists for any m while an optimal normal basis does not, enhancing the polynomial basis comparative generality. Motivated by its generality and flexibility, we choose to implement field arithmetic supported over polynomial basis. For this basis, we will describe the most important operations executed over the field, such as multiplication, reduction, squaring, and inversion.

Multiplication: The multiplication of two operands $A(x)$ and $B(x)$

$$A(x) = \sum_{i=0}^{m-1} a_i x^i,\ B(x) = \sum_{i=0}^{m-1} b_i x^i,$$

is obtained as:

$$C(x) = A(x)B(x) = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i b_j x^{i+j}. \quad (3)$$

Since the output coefficient index l depends on the values i, j such that $l = i + j$, the highest power of x in the result is $2m - 2$, which means that the result will have at most $2m - 1$ bits. This result can be reduced modulo the irreducible polynomial to the m bit representation.

Reduction: In a EC cryptosystem, an irreducible polynomial $I(x)$ with order m, usually a trinomial or pentanomial, is established to support the field [10]. It holds that $x^m \ mod\ I(x) = I(x) - x^m$, which means that multiplying a field element by x is equivalent to a left shift of its binary representation, and to sum the irreducible polynomial if the associated term to x^m is not zero. This result

allows to obtain all the values of x^k for $k > m-1$. Using one of the irreducible polynomial suggested by the National Institute of Standards and Technology (NIST) [10], namely $I(x) = x^{163} + x^7 + x^6 + x^3 + 1$ for $m = 163$, it is possible to obtain the following $(m-1) \times m$ matrix R, where the lines are the values of x^k for $m-1 < k < 2m-1$:

$$R = \begin{pmatrix} 000000000 & \dots & 000000011001001 \\ 000000000 & \dots & 000000110010010 \\ 000000000 & \dots & 000001100100100 \\ 000000000 & \dots & 000011001001000 \\ 000000000 & \dots & 000110010010000 \\ 000000000 & \dots & 001100100100000 \\ 000000000 & \dots & 011001001000000 \\ \vdots & \ddots & \vdots \\ 110010010 & \dots & 000000000000000 \\ 100100100 & \dots & 000000011001001 \\ 001001000 & \dots & 000000101011011 \\ 010010000 & \dots & 000001010110110 \\ 100100000 & \dots & 000010101101100 \\ 001000000 & \dots & 000101000010001 \\ 010000000 & \dots & 001010000100010 \end{pmatrix}$$

As an example, to determine how the reduction affects the 0 index bit of the result we go through the rightmost column of the R matrix and register the entries different from zero (the entry i correspond to the index $m+i$ of the unreduced result $C(x)$) to obtain the dependencies. In this case $d_0 = c_m + c_{(m+156)} + c_{(m+157)} + c_{(m+160)}$ can be obtained. Repeating the same procedure for the other columns, we obtain a m bit vector $D = (d_{162}, \dots, d_0)$. Adding D to the less significant m bits of $C(x)$, the reduced element is obtained. Summarizing, to reduct a polynomial to a size m field element, its representation is classified in the m least significant bits and the most significant bits, and the later set of bits is used to compute a corrective polynomial (D) that is added to the former m bits. Note that since the irreducible polynomials have very few terms different from zero, the number of dependencies is also low.

Squaring: The squaring operation is a particular case of the multiplication:

$$C(x) = A(x)^2 = A(x)A(x) = \sum_{i=0}^{m-1} a_i x^{2i}. \qquad (4)$$

From (4) it can be shown that the result coefficient $2i$ depends only on the i input coefficient. This means that the squaring operation can be obtained with a reduction operation, rewriting the input binary representation as $(a_{m-1}, 0, \dots, 0, a_1, 0, a_0)$.

Inversion: The division operation can be performed by computing the Extended Euclidean Algorithm [2]. However, this algorithm requires extra hardware, which is not appropriate for a compact implementation. An alternative is to use the Itoh-Tsujii algorithm, which computes the inversion using several squaring and $log_2(m-1) +$ $h(m-1) - 1$ multiplication operations, with $h(.)$ the Hamming weight [6]. With this algorithm, the multiplication and squaring resources can be shared to compute the inversion, suiting a compact implementation. This algorithm is based on rewriting the result of the Fermat Little Theorem which states that for any field element β, $\beta^{2^m} = \beta \Leftrightarrow \beta^{2^m-2} = \beta^{-1}$. Further details on this algorithm can be found in [6].

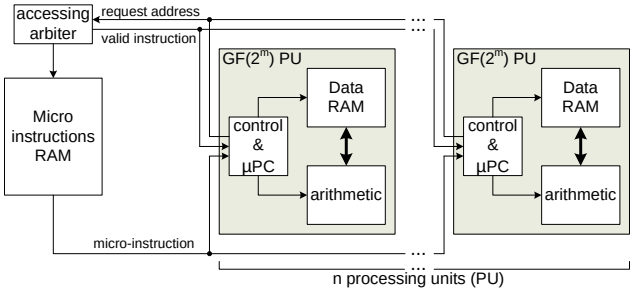

Figure 1. Processor overview.

3 Proposed Processor

The proposed architecture is targeted to a FPGA technology. The FPGA technologies provide optimized structures for data storing, namely blocks of RAM, that can be used to enhance the compactness of the design while efficiently exploring its reconfigurable capabilities. The processor overview is presented in Figure 1. There is a general control unit that is responsible for generating reset and start signals to each Processing Unit (PU), which is omitted from Figure 1 to simplify the description. The number of PUs is set by the user accordingly to the required throughput and the available hardware resources. There is an arbiter that solves the conflicts of simultaneous requests. Nevertheless, since the RAMs have true dual ports in modern FPGA technologies and the micro-instructions take several clock cycles to perform, these conflicts are expected to be minimal. When the RAM retrieves the requested micro-instruction, the arbiter is responsible to signal the destination PU, informing that the micro-instruction is valid. In the following subsection a more detailed information about the PU and the developed microcode presented.

3.1 Processing Unit

The layout of the PU can be scaled to any processing word size, field size and irreducible polynomial. In this paper 21 bit words, $GF(2^{163})$ field, and an irreducible polynomial $I(x) = x^{163} + x^7 + x^6 + x^3 + 1$ are adopted. With this configuration we divide a size m field element in 8 words of 21 bits each (index 0 to 7). The PU has a register responsi-

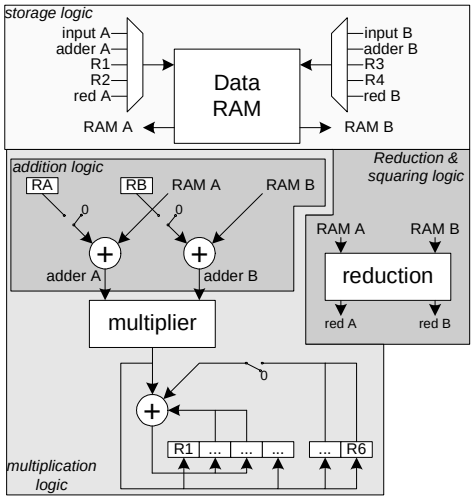

Figure 2. Processing unit architecture.

ble to hold an integer, which corresponds to the private key of the cryptosystem that is written with a dedicated micro-instruction provided by the host of the processor. The host can directly access the data RAM in the processing unit to read the results or to write data, independently of the program in the micro-instruction RAM. The flow of the program (consecutive requests and jumps) is controlled with the Micro Program Counter (μPC) in the control unit. The architecture of a PU is presented in Figure 2, depicting four different logic circuits designed to perform different functionalities.

Storage logic: This logic circuit includes a dual port RAM storing the required data, including temporary and final results. The output of the RAM is connected to the other logic sets allowing to simultaneously read or write of 2 words. The inputs come from external entities, input A and B, and from the local arithmetic units, namely the addition, reduction, and multiplication units.

Addition logic: This set is composed by two adders, two registers (RA and RB), and two controlled multiplexers that allow the adders to just pass-through data when the 0 inputs are selected. Note that the adders and the input selectors can be implemented with only a single 3 input LUT per bit, which are bounded to the 4 input LUTs in actual FPGAs. This logic is responsible for two additions of 21 bit words, namely $A + B = R_1$ and $C + D = R_2$. For this, the operands A and C are obtained from the RAM and stored in the registers RA and RB, respectively. Then the operands B and D are requested from the RAM and the results R_1 and R_2 simultaneously computed and stored in the RAM. This data flow maximizes the usage of the RAM

Table 1. Multiplication RAMs and registers agenda

Step	RAM		Registers							
	A	B	A	B	1	2	3	4	5	6
0	-	-	-	-	0	0	0	0	0	0
1	a_L	b_L	a_L	b_L	α_L	α_L	α_H	0	α_H	0
2	a_H	b_H	a_L	b_L	α_L	γ_L	γ_H	β_H	α_H	β_L
3	a_H	b_H	a_L	b_L	α_L	ω_L	ω_H	β_H	α_H	β_L

bandwidth.

Multiplication logic: The multiplication logic circuit is used to compute a 2-word$\times$2-word multiplication. This logic circuit uses the addition logic and a 3-input adder, with one of the entries selected through a multiplexer. This adder and the multiplexer can be implemented with a 4-input LUT per bit. 6 registers (R1 to R6) are used to store the temporary results. The implemented 21×21 multiplier is a completely parallel implementation derived from (3). In order to obtain the 2-word multiplication, the Karatsuba-Offman method is employed [11]. Considering two 2-word operands $A = a_H x^n + a_L$ and $B = b_H x^n + b_L$, with $n = 21$, the result C can be obtained by:

$$C = a_H b_H x^{2n} + (a_H b_L + a_L b_H) x^n + a_L b_L. \quad (5)$$

The Karatsuba-Offman method relies on the identity:

$$a_H b_L + a_L b_H = (a_H + a_L)(b_H + b_L) - a_H b_H - a_L b_L, \quad (6)$$

which allows to compute the 2-word multiplication with 3 n-bit multiplications instead of 4, obtaining sub-quadratic complexity. By noting that in $GF(2^m)$ the addition and subtraction operations are the same, and considering:

- $a_L b_L = \alpha_H x^n + \alpha_L$;
- $a_H b_H = \beta_H x^n + \beta_L$;
- $a_H b_H + a_L b_L = \gamma_H x^n + \gamma_L$;
- $(a_H + a_L)(b_H + b_L) + \gamma_H x^n + \gamma_L + \beta_L x^n + \alpha_H = \omega_H x^n + \omega_L$,

the multiplication result is given by:

$$C = \beta_H x^{3n} + \omega_H x^{2n} + \omega_L x^n + \alpha_L. \quad (7)$$

To obtain this result, the output from the RAMs and the content of the registers respect the schedule in Table 1, where a step corresponds to a clock cycle. At step 3, registers R1 to R4 contain the final result, from the least significant word in R1 to the most significant word in R4. Since at step 2, registers R1 and R4 already have the final content, it is possible to start writing 2 words of the final result to the RAM after this step. One more step is required to write the other two words in R2 and R3 to the RAM.

Reduction and squaring logic: Reduction and squaring are very similar operations, differing only in the way the input operand is provided (see Section 2.1). Therefore, to

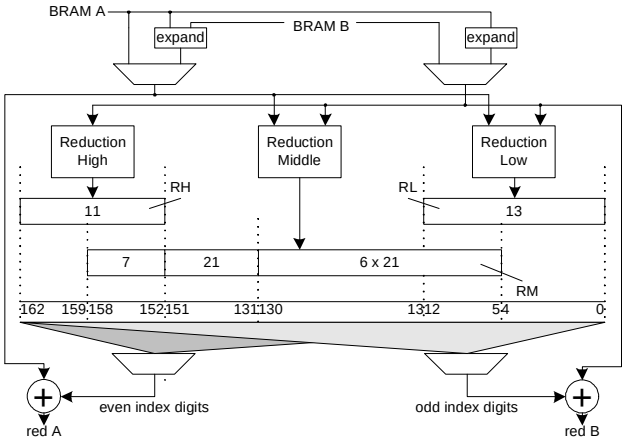

Figure 3. Reduction unit layout.

make the PUs even more compact, these two operations share resources. The reduction unit is depicted in Figure 3. The reduction matrix, for the chosen field size $m = 163$ and irreducible polynomial, coincides with the matrix R in Section 2.1. The reduction of a $2m - 1$ bit result (index 15 to 0 of 21-bit words) to a m bit result is performed in two main steps: *i)* a corrective field element is computed from the most significant bits of the unreduced element (bits m to $2m - 2$ bits), and *ii)* this corrective element is added to the least significant bits of this element (bits 0 to $m - 1$).

To obtain the corrective factor, the $m - 1$ most significant bits should be regarded in the unreduced stored data. Since 8 words correspond to 168 bits, there are 5 bits to be reduced in the word index 7. The Reduction Low unit in Figure 3 computes the reduction that results from these 5 bits and from the other dependencies which intervene in the lower part of the corrective factor, corresponding to the dependencies defined by the rightmost columns of the matrix R in Section 2.1. The Reduction High unit computes the reduction from the most significant word (index 15) of the unreduced input, with the dependencies defined by the leftmost columns of matrix R. The output of the Reduction Low and Reduction High units are stored in the RL and RH registers, respectively, as Figure 3 suggests. Considering w the word size, n the number of bits of the irreducible polynomial excluding the m order bit, $I'(x) = x^7+x^6+x^3+1$, and the possibility of simultaneously accessing two words from the RAM, we define a $2w \times (2w + n - 1)$ matrix R' equal to the up-right corner of matrix R. Moving the matrix R' through the diagonal in direction to the bottom-left corner of the matrix R, both matrices will coincide until the R' matrix achieves the leftmost column of matrix R. This property allows us to construct a smaller reduction unit, with only the matrix R' dependencies, and reuse this unit for the remaining pair of words that need to be reduced (which correspond to the central columns of matrix R). In this particular case we are using $w = 21$ and $n = 8$. The remaining words to reduce correspond to index 8 to 14. Since the RAM outputs 2 words per clock cycle we can reduce $2w = 42$ bits at once. Note that from the reduction of two words results $2w + n - 1$ bits. The extra $n - 1 = 7$ bits must be added to the following reduction of two more words. The Reduction Middle unit in Figure 3 is responsible for performing this reduction using the R' matrix. The extra 7 bits are stored in the most significant positions of the register RM at the final of a 2-word reduction. Excluding these 7 bits, RM is a $2w$ positions right shift register that stores the consecutive $2w$ bit result of the reduction unit. Note that, in the case of this work, an odd number of words need to be managed, thus in the last Reduction Middle operation, one of the words is set to zero.

With the values of RH, RM, and RL registers updated with the partial reduction results, the contents of these registers are added by aligning the registers as depicted in Figure 3. Then the least significant words of the operand to be reduced are browsed through the BRAM and added with the correspondent even or odd words of the corrective element. The most significant word result is masked in order to remove the extra 5 bits which correspond to an index higher than $m - 1 = 162$.

To obtain the squaring operation from this unit, two Expand units are depicted in Figure 3, for the expansion (rewriting) of the output of the BRAM, with the odd index bits equal to zero and the i even index bits equal to the $i/2$ index bits (see Section 2.1).

3.2 Microcode

The microcode adopted for the proposed processor can be classified into two main micro-instruction types. The complex micro-instructions (type I) are performed over field elements, while the lower complexity micro-instructions (type II) operate over words. There is a type I reserved micro.instruction that corresponds to a personalized sequence of type II operations. Each micro-instruction is coded in 32 bits as depicted in Figure 4, where Op.X specifies direct addresses in the data or instruction RAMs, and Wi specifies direct or Op.X relative addresses depending on the Ci information.

Type I instructions are used to access the memory (`READ` and `WRITE`) and the key register (`KEY`), to compute the m bit add, squaring and reduction operations (`ADD`, `SQR` and `RED`), and to control the flow by conditionally jumping to a micro-instruction address depending of the key register or by turning the PU to an idle state (`JMP` and `END`). A customizable instruction (`PERS`) is also reserved, corresponding to an user defined sequence of type II instructions. The type II instructions allow to add and multiply 2-word operands (`eADD` and `eMULT`). An instruction determines

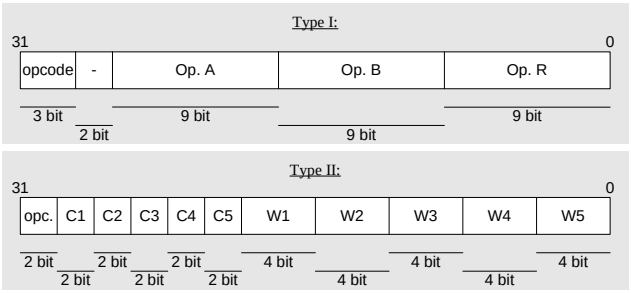

Figure 4. Micro-instruction format.

the end of a type II sequence (eRET) and, consequently, the end of the PERS type I instruction.

4. An illustrative application

The proposed system was tested using real elliptic curve point operations. In particular, herein we present the case of a point multiplication supported by a personalized field multiplication Karatsuba-Offman algorithm. This latter algorithm is supported by the properties described in Section 3.1. The 2-word algorithm is recursively applied to obtain a 8-word algorithm. Three 4-word multiplications are combined in order to obtain the 8-word multiplication. To obtain each 4-word multiplication, three 2-word multiplications have to be combined. Since we have 3 recursive levels and 3 multiplications per recursive call, to construct this algorithm nine (3×3) 2-word multiplication operations are required. The multiplications results are combined together with several addition operations to achieve the complete 8-word multiplication operation.

The implemented point multiplication algorithm has lead to the fastest point multiplications in the literature [3] for FPGAs. This algorithm, based on the Montgomery multiplication algorithm, was originally proposed in [8] and allows to compute the resulting x coordinate using only the input x coordinate. A standard projective representation was suggested for this point multiplication algorithm,which maps an affine representation (x, y) to the projective representation (X, Y, Z) as $x \longleftarrow X/Z$ and $y \longleftarrow Y/Z$. The conversion from affine to the projective representation is straightforward, since $Z = 1$ is considered. The conversion from projective to affine requires the inversion of Z, because the final value of Z can be different from 1. The implemented point multiplication P by a scalar (key) s is described in Algorithm 1. The routines Madd and Mdouble compute the projective version of the addition (X_A, Z_A) and doubling (X_D, Z_D) operations, respectively, as:

$$\begin{cases} X_D = X_1^4 + bZ_1^4 \\ Z_D = X_1^2 Z_1^2 \end{cases} ; \begin{cases} Z_A = (X_1 Z_2 + X_2 Z_1)^2 \\ X_A = xZ_A + X_1 Z_2 X_2 Z_1 \end{cases} \quad (8)$$

Algorithm 1 Point Multiplication Algorithm

Require: $s := (s_{n-1} s_{n-2} \dots s_0)$ with $s_{n-1} = 1$
$P = (x, y) \in GF(2^m) \times GF(2^m)$;
Ensure: $Q = kP$;
$X_1 := x$; $Z_1 := 1$; $X_2 := x^4 + b$; $Z_2 := x^2$;
for $i = n - 2$ downto 0 **do**
if $k_i = 1$ **then**
(X_1, Z_1)=Madd(X_1, Z_1, X_2, Z_2);
(X_2, Z_2)=Mdouble(X_2, Z_2);
else
(X_2, Z_2)=Madd(X_2, Z_2, X_1, Z_1);
(X_1, Z_1)=Mdouble(X_1, Z_1);
end if
end for
return Q :=Mxy(X_1, Z_1, X_2, Z_2);

The Mxy routine is responsible for converting the projective coordinates to affine coordinates. This routine was adapted in [1] to manage the compact representation proposed in [12], which consist in representing an EC point by its x coordinate and the bit $T(y/x)$, where $T(.)$ is the trace operator. This trace operator can be very efficiently computed when using polynomial basis by an exclusive or of fixed element's bits. The number of bits which intervene in the trace operator are only two for this implementation field characteristics. The result $T(y_R/x_R)$ can be obtained from the input $T(y/x)$ as [1]:

$$T\left(\frac{y_R}{x_R}\right) = T\left(\frac{y}{x}\right) + T\left(\frac{(xZ_1 + X_1)\,(x(X_1 Z_2 + X_2 Z_1) + X_1 X_2)}{x X_1 Z_1 Z_2}\right). \quad (9)$$

The Mxy routine combines the computation of the resulting x coordinate $x = X_1/Z_1$ and the input of the second term trace operator in (9). The computation of the trace operation in (9) can be easily performed by the host in order to obtain the compact representation whenever data has to be transmitted.

5. Results and related work

The proposed processor supported by one PU was implemented and thoroughly tested in a Xilinx Virtex 4 FPGA platform (XC4VSX35). The FPGA programming file was obtained from a VHDL description of the processor synthesized with Synplify Premier version 9.6.2 tools. The micro-instruction and data memories were configured using the Xilinx Coregen version 10.1 tool. The Place&Route was performed with the ISE version 10.1 tools. In order to compare the design characteristics with the related art, the processor was also implemented in a Xilinx VirtexE technology (XCV1000E). The results for the two target technologies are in Table 2.

Table 2. Placed and Route results.

device	Area	Freq.	# BRAMs
Virtex 4	1095 slices	150 MHz	2
Virtex E	1101 slices	82 MHz	7

Table 3. Micro-instructions latency in clock cycles.

SQR/RED	ADD	JMP	PERS	END
14	13	3	3*	3

eMULT	eADD	eRET
5	3	3

Note that the existent BRAMs in the different technologies have different sizes. In the Virtex E technology, the existing BRAM have 4096 bit positions, accessed in words up to 16 bit, allowing for the data (512×21-bit locations) and micro-instruction (512×32-bit locations) RAMs to be implemented with 3 and 4 BRAMs, respectively. In the Virtex 4 technology each BRAM has 16 kbit positions, which allows to map 512 positions of 32 bits in a single embedded block. With the proposed implementation, the micro-instructions used to implement the cryptographic procedures have the latency presented in Table 3.

For the Karatsuba-Offman multiplication algorithm described in (Section 4) to be computed in a PERS micro-instruction, 9 eMULT, 40 eADD, and 1 eRET micro-instructions are required, corresponding to 171 clock cycles (including the 3 clock cycles for the call). In the point multiplication Algorithm 1, the inversion algorithm consists of 9 PERS (field multiplication), 9 RED, and 162 SQR operations, resulting in a $3,933$ clock cycles latency. With this microcoded inversion operation we are able to implement the complete point multiplication algorithm. This algorithm can be divided into three parts: i) an initialization step before the main loop, ii) the main cycle, and iii) the custom conversion procedure to support the EC points compact representation. The complete algorithm consists of 988 PERS, 988 RED, 654 SQR, 493 ADD, 164 JMP, 1 END, and 1 microcoded inversion, resulting in a $202,773$ clock cycles latency. The total time required for one point multiplication depends on the clock frequency. In Table 4 the computation characteristics of the proposed processor is evaluated and accessed relatively to three other EC processors [4, 6, 3]. In [4] and [6], processors supported in microcode are proposed, while in [3] a dedicated pipeline architecture is proposed controlled by application specific instructions.

In the implementation proposed in [4] a normal basis is used to support the field, using a Massey-Omura multiplier. This processor is controlled with three micro-instructions to multiply, add, and square field elements. The inversion algorithm is identical to the one used in this work. The consecutive micro-instructions are provided from outside the device, thus the same program has to be repeatedly provided even if the same program is computed several times. This is a clear disadvantage regarding the processor herein proposed where the program is internally stored and the loops are unrolled. Nevertheless, the processor presented in [4] presents 2.39 more area obtaining a slices$\times$time metric of 3055, which is 12% higher than the one herein proposed. Our processor has a better balance between area and computation time. This comparison is a estimation, since different technologies and slightly different field sizes are used.

*The personalized type-II program latency has to be added to this value.

Table 4. Implementation Results for the EC point multiplication processors. (†7-depth pipeline specialized processor)

Ref.	Device	m	Freq. [MHz]	Area [slices]	Time [ms]
[4]	XC4085XLA	191	32	$2,634$	1.160
[6]	XCV1000	163	-	$1,945$	9.450
Ours	XCV1000E	163	82	$1,101$	2.472
[3]	XC4VLX200	163	154	$16,209$	$0.020^{\dagger}$
Ours	XC4VSX35	163	150	$1,095$	1.351

In [6] an optimal normal basis is used, supporting a Massey-Omura multiplier. The authors provide information that allows us to estimate the area and latency for the field used in our implementation. Note that there is no optimal basis for the field used in our implementation, thus the results presented in Table 4 are estimated. The area estimation was obtained considering a linear increase with the field size, as suggested by the authors. The latency for $GF(2^{163})$ was estimated from three entries of a latency table fitted by a curve. Comparing this processor with the one herein proposed an area gain of 76% and a performance gain of 282% can be achieved. The obtained slices$\times$time metric for this implementation is $18,380$, which is more than 6 times higher than ours. Thus, the design herein proposed is more efficient.

While presenting a competitive performance both in terms of time and area, our processor has showed to be more flexible than the structures proposed in [6] and [4], since the personalization of the operations, namely the field multiplication, can be performed. In order to fully analyze the performance of the EC processor herein proposed, it was also compared with a fully dedicated structure, namely the one proposed in [3]. This processor implements the same point multiplication algorithm supported by polynomial basis, and is, from the authors knowledge, the fastest proposed in the literature. The pipelining and high parallelization result in a very high throughput. Our processor has the advantage of being able to compute the used algorithm as well as different ones while using almost 15 times less area, being a significantly more compact solution. The approach in [3] is

not expected to result in compact solutions, since this implementation performance is supported by minimal routing delays which will increase with the data reutilization required for compact implementations. Although the solution in [3] to be controlled with application-specific instructions, the architecture is constructed for an already known instruction sequence. Therefore, this solution is very similar to a dedicated processor.

Regarding all the three related art implementations we have at most half the area, providing adaptability by scaling the logic inside the processing unit and also by replicating the processing units to enhance the performance.

In order to fully test the proposed architecture, a prototype of a system based on the proposed processor was implemented on a Annapolis Wildcard 4 prototyping board. This board contains a XC4VSX35 FPGA device, PCI communication logic and clock programmability logic, all accessed from a host computer Application Programming Interface (API). The PCI communication logic from the FPGA device consists of registers that are accessed from both sides. With this, the host computer can provide the general control to the system, as the start computation order, and the data to be computed. This implementation consists of a processor with 4 PUs (2 PUs per instruction BRAM port). The occupied area is 4,146 slices (almost 4 times the area for a single PU) and the working frequency is of 147 MHz, very close to the frequency obtained for a single PU (an expected result since the critical path is inside the PUs). By computing the same EC point multiplication algorithm by the 2 PU that share the same port of the instruction BRAM, the number of collisions in the requests solved by the arbiter is 0.37% of the total clock cycles where collisions may occur, when the 2 PU are operating simultaneosly with random starting points.

6. Conclusions and future work

A very compact and adaptable processor for cryptographic applications supported over $GF(2^m)$ is proposed in this paper. This processor has a specific microcode with two complexity levels, allowing the user to perform custom cryptographic algorithms and to personalize the field operation. This flexibility is further enhanced by FPGA reconfiguration capabilities that allow to scale the dedicated units and to replicate the processing units.

This processor is capable of computing EC point multiplication in $1.35\ ms$ using $1,095$ slices. Results suggest that the required area is at most half of the related state of the art implementations. The use of polynomial basis, instead of the normal basis used in the related art microcoded implementations, allow to better suit different personalized operations. Comparing with dedicated and pipelined implementations, our processor can achieve 16 times less area while maintaining a high throughput.

This compact implementation is well fitted for application in devices with constrained resources where the devices' specific computation has to co-exist with the cryptographic procedures, such as, sensor networks nodes.

Future work will pass by the automatic scaling of the proposed cryptographic processor, in order to accelerate the customization for different field sizes. With this and the use of dynamic partial reconfiguration, we expect to achieve an automatically adaptable processor with high area/performance efficiency.

6.1 Processor evaluation prototypes

Processor evaluation prototypes are available for download at *http://sips.inesc-id.pt/˜sfan/prototypes/microEC*.

References

[1] S. Antão, R. Chaves, and L. Sousa. Efficient Ellitpic Curve Processor over GF(2^m). *International Conference on Field-Programmable Technology 2008, Proceedings*, December 2008.

[2] H. Brunner, A. Curiger, and M. Hofstetter. On computing multiplicative inverses in GF(2^m). *IEEE Transactions on Computers*, 42(8):1010–1015, August 1993.

[3] W. Chelton and M. Benaissa. Fast Elliptic Curve Cryptography on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(2):198–205, February 2008.

[4] M. Ernst, B. Henhapl, S. Klupsch, and S. Huss. FPGA based hardware acceleration for elliptic curve public key cryptosystems. *The Journal of Systems & Software*, 70(3):299–313, 2004.

[5] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[6] P. Leong and I. Leung. A microcoded elliptic curve processor using FPGA technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(5):550–559, 2002.

[7] K. Leung, K. Ma, W. Wong, and P. Leong. FPGA implementation of a microcoded elliptic curve cryptographicprocessor. *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 68–76, 2000.

[8] J. Lopez and R. Dahab. Fast Multiplication on Elliptic Curves over GF(2^m) without Precomputation. *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES 99, Proceedings*, 1717:316–327, August 1999.

[9] V. Miller. Uses of elliptic curves in cryptography, Advances in Cryptology, CRYPTO 1985. *Lecture Notes in Computer Science*, 218:417–426, 1986.

[10] F. NIST. 186-2, Digital Signature Standard (DSS), January 2000.

[11] N. Saqib, F. Rodriguez-Henriquez, and A. Diaz-Perez. A parallel architecture for fast computation of elliptic curve scalar multiplication over GF(2^m). *18th International Parallel and Distributed Processing Symposium, 2004, Proceedings*, April 2004.

[12] G. Seroussi. Compact Representation of Elliptic Curve Points over F_{2^n}. *HP Laboratories Technical Report*, September 1998.