Hardware Implementation of an Elliptic Curve Processor over $GF(p)$

Sıddıka Berna Örs[1], Lejla Batina[1,2], Bart Preneel[1], Joos Vandewalle[1]
[1]Katholieke Universiteit Leuven, ESAT/SCD-COSIC
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
{sbors, Lejla.Batina, Bart.Preneel, Joos.Vandewalle}@esat.kuleuven.ac.be
[2]SafeNet BV
Boxtelseweg 26a, 5261 NE Vught, The Netherlands

Abstract

This paper describes a hardware implementation of an arithmetic processor which is efficient for bit-lengths suitable for both commonly used types of Public Key Cryptography (PKC), i.e., Elliptic Curve (EC) and RSA Cryptosystems. Montgomery modular multiplication in a systolic array architecture is used for modular multiplication. The processor consists of special operational blocks for Montgomery Modular Multiplication, modular addition/subtraction, EC Point doubling/addition, modular multiplicative inversion, EC point multiplier, projective to affine coordinates conversion and Montgomery to normal representation conversion.

Keywords: *Elliptic Curve Cryptosystems, Modular Operations, FPGA*

1 Introduction

Elliptic Curve Cryptography (ECC) was proposed independently by Miller [13] and Koblitz [7] in the 80's. Since then a considerable amount of research has been performed on secure and efficient ECC implementations. The benefits of ECC, when compared with classical cryptosystems such as RSA [19], include: higher speed, lower power consumption and smaller certificates, which are especially useful for wireless applications.

The performance of an elliptic curve cryptosystem and of other public key cryptosystems, is mostly determined by the efficient implementation of finite field arithmetic. In this work a hardware architecture of a processor for ECC over finite field $GF(p)$ is presented. The most critical operation for latency is modular multiplication. We use our systolic array multiplier based on Montgomery's Modular Multiplication (MMM) algorithm [14] which is proposed in [16]; this multiplier is proven to be very efficient for modular exponentiation as the basic operation for RSA cryptosystems [1].

The processor consists of special operational blocks for MMM, modular addition/subtraction (MAS), EC point doubling/addition, modular multiplicative inversion, EC point multiplier, projective to affine coordinates conversion and Montgomery to normal representation conversion. Hence it can be programmed by the host to execute any of these operations in any order. It is possible to use the proposed processor not only for ECC, but also for any system that modular arithmetic operations are essential for, such as the RSA cryptosystem.

The basic operations are MMM and MAS. The other blocks include a finite state machines (FSMs) which controls the execution of these operations in the right order. The critical path depends only on the critical path of circuits for MMM and MAS. The architecture of these blocks is designed to ensure a short critical path to allow for high clock frequencies which are independent from bit-length of the parameters of ECC. For simplicity, all blocks were designed separately with their own FSMs. This allows for independent optimization and testing of the building blocks.

The remainder of this paper is organized as follows. In Section 2 we discuss the related work. Section 3 provides the mathematical background for Montgomery Multiplication Method (MMM) and ECC over $GF(p)$. Section 4 describes the hardware implementation; some details are omitted due to space limitation. Section 5 concludes the paper.

2 Previous Work

To the best of our knowledge, the first documented ECC processor over fields $GF(p)$ is proposed by Orlando and Paar [15]. The Elliptic Curve Processor (ECP) is scalable in terms of area and speed and especially suited for FPGAs. The authors estimate that it would take 3 ms to compute one 192-bit point multiplication. However, this superb timing was estimated by assuming 100% throughput from the multiplier. The expected latency was not considered. Their multiplier is also based on the MMM algorithm but it is a generalized version with quotient pipelining introduced by Orup in [17]. We use the basic MMM algorithm from which we only exclude the modular reduction as a result of the bound adjustment. In this way no pre-computation is required which results substantial memory saving. Their multiplier has a semi-systolic architecture while the multiplier presented here is fully systolic. This results in an important flexibility which is unrelated to any specific parameter choice. Orlando and Paar also used an adaptation of a fixed base exponentiation method as introduced by Brickell *et al.* in [3]. This algorithm is assumed to be 4 times faster than standard double-and-add algorithm which is used here. However, it involves a known point calculation which is a limiting factor with respect to various applications of ECC.

Wolkerstorfer proposes a dual-field arithmetic unit that offers all instructions required for both types of finite fields: $GF(p)$ and $GF(2^m)$ in [22]. He uses a redundant number representation and a special multiplication with interleaved modular reduction. Inversion is performed by the Extended Euclidean Algorithm. This is a low-power architecture that can be realized on moderate silicon area; the author claims that it requires just a little more hardware resources than for a pure $GF(p)$ multiplier.

Goodman and Chandrakasan proposed a domain-specific reconfigurable cryptographic processor (DSRCP) in [6]. The instruction set definition of the DSRCP was dictated by the IEEE 1363 Public Key Cryptography Standard document. A list of the arithmetic functions required to implement the various primitives defined in the standard was tabulated in a functional matrix, which was then used to define the instruction set architecture (ISA) of the processor. The ISA contains 24 instructions broken up into six types of operations: conventional arithmetic, modular integer arithmetic, GF arithmetic, elliptic curve field arithmetic over GF, register manipulation and processor configuration.

3 Mathematical background

3.1 Elliptic curves over $GF(p)$

An elliptic curve E is often expressed in terms of the Weierstrass equation: $y^2 = x^3+ax+b$, where $a, b \in GF(p)$ with $4a^3+27b^2 \neq 0 \pmod{p}$. The inverse of the point $P = (x_1, y_1)$ is $-P = (x_1, -y_1)$. The sum $P + Q$ of the points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ (assume that $P, Q \neq \mathcal{O}$, and $P \neq \pm Q$) is the point $R = (x_3, y_3)$ where: $\lambda = \frac{y_2-y_1}{x_2-x_1}$, $x_3 = \lambda^2 - x_1 - x_2$, $y_3 = (x_1 - x_3)\lambda - y_1$.

For $P = Q$, the "doubling" formulae are: $\lambda = \frac{3x_1^2+a}{2y_1}$, $x_3 = \lambda^2 - 2x_1$, $y_3 = (x_1 - x_3)\lambda - y_1$.

The point at infinity $\mathcal{O}$ plays a role analogous to that of the number 0 in ordinary addition. Thus, $P + \mathcal{O} = P$ and $P + (-P) = \mathcal{O}$ for all points P. The points on elliptic curve together with the operation of "addition" form an abelian group. Then it is straightforward to introduce the point or scalar multiplication as main operation for ECC. This operation can be calculated by using double-and-add algorithm as shown in Algorithm 1. For details see [13, 7, 2].

Algorithm 1 Elliptic Curve Point Multiplication

```
Require: EC point $P = (x, y)$, integer $k$, $0 < k < M$, $k = (k_{l-1}, k_{l-2}, \cdots, k_0)_2$,
    $k_{l-1} = 1$ and $M$
Ensure: $Q = (x', y')$
 1: $Q \leftarrow P$
 2: for $i$ from $l - 2$ downto 0 do
 3:     $Q \leftarrow 2Q$
 4:     if $k_i = 1$ then
 5:         $Q \leftarrow Q + P$
 6:     end if
 7: end for
```

In the above definition of EC group affine coordinates are used, but so-called projective coordinates have some implementation advantages. The point addition can be done in projective coordinates using almost only field multiplications. Only one inversion is needed at the end of a point multiplication operation. We have used the modified Jacobian (J^m) coordinates as proposed by Cohen *et al.* in [5] because EC point doubling is fastest in this representation. They represent internally the Jacobian coordinates as a quadruple $\left(X, Y, Z, aZ^4\right)$. This representation is called modified Jacobian coordinate system and denoted by the authors as J^m. The algorithms for EC point addition and doubling are as follows [5].

Let $P = \left(X_1, Y_1, Z_1, aZ_1^4\right)$, $Q = \left(X_2, Y_2, Z_2, aZ_2^4\right)$ and $P + Q = R = \left(X_3, Y_3, Z_3, aZ_3^4\right)$. The addition formulas in J^m are the following ($P \neq \pm Q$).

$$\begin{array}{l} U_1 = X_1Z_2^2,\ U_2 = X_2Z_1^2,\ S_1 = Y_1Z_2^3,\ S_2 = Y_2Z_1^3,\ H = U_2 - U_1,\ r = S_2 - S_1 \\ X_3 = -H^3 - 2U_1H^2 + r^2,\ Y_3 = -S_1H^3 + r\left(U_1H^2 - X_3\right),\ Z_3 = Z_1Z_2H,\ aZ_3^4 = aZ_3^4 \end{array} \tag{1}$$

The doubling formulas in J^m are the following ($R = 2P$).

$$\begin{array}{l} S = 4X_1Y_1^2,\ U = 8Y_1^4,\ M = 3X_1^2 + \left(aZ_1^4\right) \\ X_3 = -2S + M^2,\ Y_3 = M(S - X_3) - U,\ Z_3 = 2Y_1Z_1,\ aZ_3^4 = 2U\left(aZ_1^4\right) \end{array} \tag{2}$$

3.2 Montgomery Modular Multiplication

The Montgomery product is defined as: $Mont(x, y) = xyR^{-1} \mod N$, where $N = (n_{l-1} \cdots n_1 n_0)_b$, $0 \leq x,\ y < N$, $R = b^l$, $b = 2^\alpha$ with $gcd(N, b) = 1$.

Montgomery's method for multiplying two integers x and y (called N-residues) modulo N, avoids trial division by N which is the most expensive operation in hardware. The Montgomery representation of $x \in \mathbb{Z}_\mathbb{N}$ is $xR \mod N$ and it allows very efficient modular arithmetic especially for multiplication [14].

The original proposal of Montgomery had a conditional subtraction included at the end of the algorithm. For efficiency as well as resistance against side-channel attacks [9, 10] a bound for R is given as $4N < R$ to avoid this subtraction by Walter in [21]. This bound guarantees that for inputs $X, Y < 2N$ the output is also bounded by $T < 2N$.

We will take $\alpha = 1$ for simplicity and make the iteration starting from Step 2 execute $l+2$ times instead of l times as in the original proposal. By these changes the desired bound is achieved as $4N < R = 2^{l+2}$. Algorithm 2 is the algorithm for Montgomery modular multiplication without final subtraction which has the properties given above.

Algorithm 2 Montgomery modular multiplication without final subtraction

```
Require: Integers $N = (n_{l-1} \cdots n_1 n_0)_2$, $x = (x_l \cdots x_1 x_0)_2$, $y = (y_l \cdots y_1 y_0)_2$ with $x \in [0, 2N-1], y \in$
    $[0, 2N-1]$, $R = 2^{l+2}$, $gcd(N, 2) = 1$ and $N' = -N^{-1} \mod 2$ (Notation $T = (t_{l+1} t_l ... t_0)$)
Ensure: $T = xyR^{-1} \mod 2N$
1: $T \leftarrow 0$
2: for $i$ from 0 to $l+1$ do
3:     $m_i \leftarrow t_0 \oplus x_i y_0$
4:     $T \leftarrow (T + x_i y + m_i N)/2$
5: end for
```

All the operations will be done modulo $2N$ through EC point multiplication. The last step is to convert the Montgomery representation of the coordinates of the resulting point back to the normal representation. This is done by calculating the Montgomery modular multiplication of the coordinates and 1, $Mont(xR, 1) = xRR^{-1} = x$. It can be easily proved that $Mont(T, 1) \leq N$, if $0 \leq T < 2N$.

4 Hardware Implementation

Our Elliptic Curve processor (ECP) can be divided into 5 levels hierarchically as shown in Fig. 1.

The operation blocks on each level from top to bottom are as follows:

- **Level 1:** Main Controller (MC)
- **Level 2:**
    1. Affine to projective coordinates converter (AtoP): $(x, y) \rightarrow (X, Y, Z, aZ^4)$ such that $X = x$, $Y = y$, $Z = 1$ and $aZ^4 = a$
    2. Normal to Montgomery representation converter (NtoM)
    3. EC point multiplier (EPM)
    4. Projective to affine coordinates converter (PtoA)
    5. Montgomery to normal representation converter (MtoN)

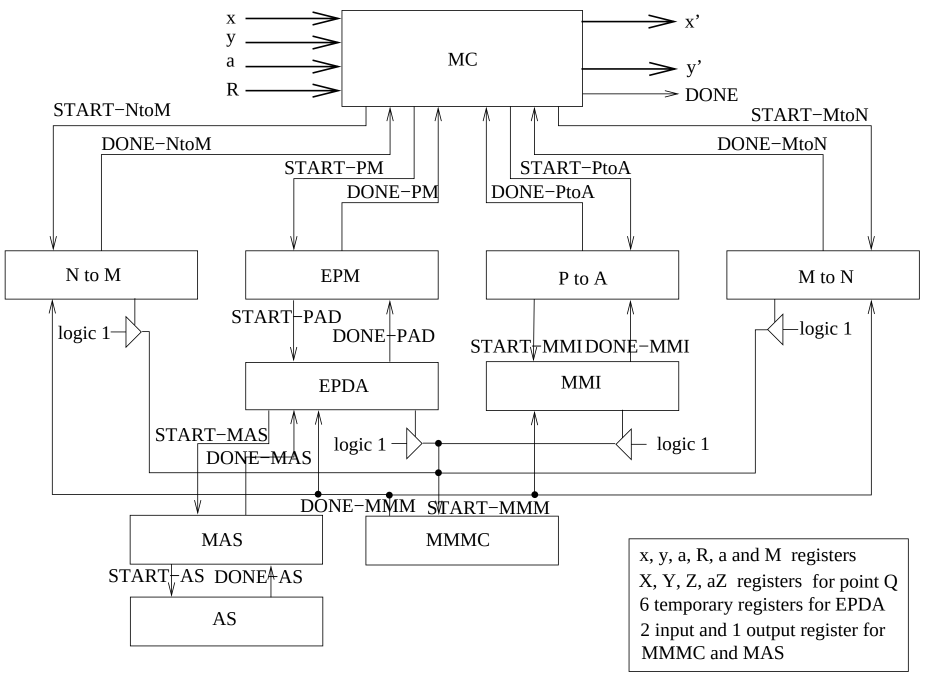

Figure 1. EC point multiplier circuit block diagram

- **Level 3:**
  1. EC Point doubling, addition circuit (EPDA)
  2. Modular Multiplicative Inverter (MMI)
- **Level 4:**
  1. Montgomery Modular Multiplication Circuit (MMMC)
  2. Modular Addition, Substraction circuit (MASC)
- **Level 5:** Addition, Substraction circuit (ASC)

For simplicity all blocks were designed separately with their own FSMs and data paths. This allows for independent optimization and testing of the building blocks. The VHDL code was written by describing the bit-length N of the coordinates x and y of P and the bit-length l of k as parameters. So this design is suitable for any N and l. In the following sections we have described the system using a top-down approach.

4.1 Main Controller

MC includes a FSM with 5 states. The algorithmic state machine (ASM) chart [11] of MC is shown in Fig. 2.(a). The START signal is the instruction signal from host. MC instructs, NtoM to start conversion from normal to Montgomery representation, EPM to start point multiplication, PtoA to start conversion from projective to affine coordinates and MtoN to start a conversion from Montgomery to normal representation one after another by setting START-NtoM, START-PM, START-PtoA and START-MtoN signals, respectively. The DONE-NtoM, DONE-PM, DONE-PtoA and DONE-MtoN signals indicate that the related operations are finished. The DONE signal indicates to the host that a complete point multiplication operation is finished and the results are ready on output ports.

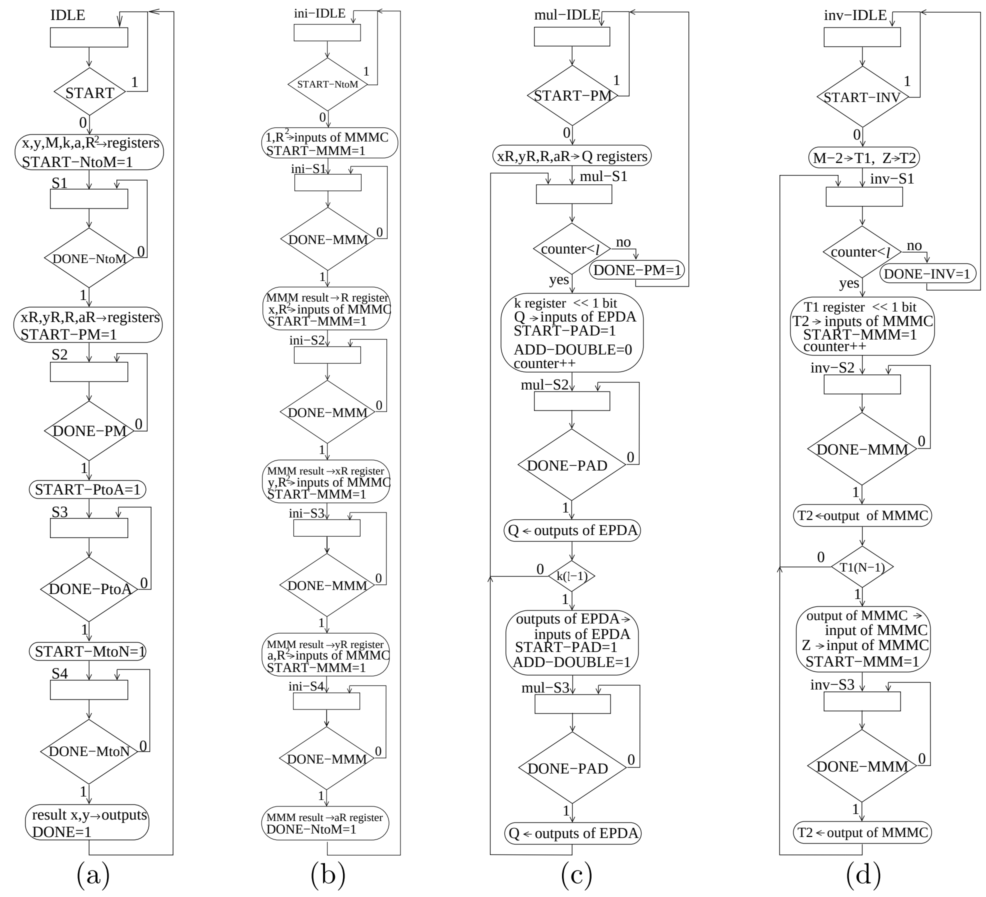

Figure 2. ASM charts of the operational blocks: (a) MC, (b) NtoM, (c) EPM, (d) MMI

4.2 Normal to Montgomery representation converter

The conversion of an integer x from the normal representation to the Montgomery representation is done as $Mont(x, R^2) = xR^2R^{-1} \bmod M = xR \bmod M$. Multiplication by MMMC of two numbers that are in Montgomery representation will produce the Montgomery representation of product as $Mont(xR, yR) = xRyRR^{-1} \bmod M = xyR \bmod M$. Modular addition and subtraction of two numbers that are in Montgomery representation will produce the Montgomery representation of the sum or difference as $xR \bmod M \pm yR \bmod M = (x \pm y)R \bmod M$. Because of these relations; the Montgomery representation of the coordinates of P, the coefficient a and number 1 will be calculated in the beginning of point multiplication by the NtoM circuit and all the operations during the EC point multiplication will be done in Montgomery representation.

NtoM includes a FSM with 5 states. The ASM chart of NtoM is shown in Fig. 2.(b). NtoM waits in first (ini-IDLE) state until the START-NtoM signal from MC is set. NtoM makes MMMC to execute 4 MMMs, $Mont(1, R^2) = R \bmod M$, $Mont(x, R^2) = xR \bmod M$, $Mont(y, R^2) = yR \bmod M$, $Mont(a, R^2) = aR \bmod M$. After DONE-MMM is set in last state, NtoM sets DONE-NtoM signal and goes back to (ini-IDLE) state.

4.3 EC Point Multiplier

EPM includes a FSM with 4 states to control the execution of Algorithm 1. The ASM chart of EPM is shown in Fig. 2.(c). The circuit stays in first (mul-IDLE) state until the START-PM signal from the MC is set. DONE-PM signal indicates that the scanning of the bits of k is finished, so the result of the operation can be read from the output ports. EPM instructs EPDA to start a point double operation by setting START-PAD signal and resetting ADD-DOUBLE signal and a point addition operation by setting START-PAD and ADD-DOUBLE signals. DONE-PAD from EPDA indicates the a point double or addition operation is finished.

4.4 Projective to affine coordinates converter

After finishing the EC point multiplication the result point Q must be converted from J^m coordinates to affine coordinates. This is done as $\left(X, Y, Z, aZ^4\right) \rightarrow (x, y)$ such that $x = XZ^{-2}$ and $y = YZ^{-3}$ [5].

PtoA includes a FSM with 6 states to control above operations. PtoA waits in first (PtoA-IDLE) state until the signal START-PtoA from MC is set. After it is set, PtoA visits the other five states in the following order and after DONE-MMM signal from MMM circuit is set in (PtoA-S5) state, PtoA sets DONE-PtoA signal and goes back to (PtoA)-IDLE state.

- PtoA-S1: $Z^{-1}R$ =Modular Multiplicative Inversion of Z
- PtoA-S2: $Z^{-2}R = Mont(Z^{-1}R, Z^{-1}R)$
- PtoA-S3: $xR = XZ^{-2}R = Mont(XR, Z^{-2}R)$
- PtoA-S4: $Z^{-3}R = Mont(Z^{-1}R, Z^{-2}R)$
- PtoA-S5: $yR = YZ^{-3}R = Mont(YR, Z^{-3}R)$

4.5 Montgomery to normal representation converter

Because the coordinates of the product point must be in normal representation, as a last action a conversion from Montgomery representation to normal representation is needed. This conversion requires two additional execution of the MMM operation with the inputs xR and 1, then yR and 1, as $x = Mont(xR, 1) = xRR^{-1}$, $y = Mont(yR, 1) = yRR^{-1}$.

4.6 EC Point doubling, addition

When we convert the input point P from affine coordinates to projective coordinates we take Z as 1. The J^m representation of $P(x, y)$ is $(x, y, 1, a)$. During the execution of point multiplication one of the points to be added is always P. According to these properties we can take $Z_1 = 1$ for EC point addition. Because there are both MMMC and modular addition/subtraction (MAS) circuits available, these operations can be executed in parallel. EC point addition and doubling can be realized by Algorithm 3.(a). and (b)., respectively.

Fourteen states and six temporary registers are needed for EC point addition and also for EC point doubling. Because completing one MAS operation takes shorter time than one MMM, the latency of one state is the same as one MMM. Hence the total execution time of EC point addition is $14T_{MMM}$, with T_{MMM} latency of one MMM. The total execution time of EC point doubling is $8T_{MMM} + 6T_{MAS}$, with T_{MAS} latency of one MAS.

Algorithm 3 EC point addition and doubling

```
(a)
Require: $P_1 = (x, y, 1, a)$, $P_2 = (X_2, Y_2, Z_2, aZ_2^4)$
Ensure: $P_1 + P_2 = P_3 = (X_3, Y_3, Z_3, aZ_3^4)$
  1.  $T_1 \leftarrow Z_2^2$
  2.  $T_2 \leftarrow xT_1$
  3.  $T_1 \leftarrow T_1 Z_2$                $T_3 \leftarrow X_2 - T_2$
  4.  $T_1 \leftarrow yT_1$
  5.  $T_4 \leftarrow T_3^2$                  $T_5 \leftarrow Y_2 - T_1$
  6.  $T_2 \leftarrow T_2 T_4$
  7.  $T_4 \leftarrow T_4 T_3$                $T_6 \leftarrow 2T_2$
  8.  $Z_3 \leftarrow Z_2 T_3$                $T_6 \leftarrow T_4 + T_6$
  9.  $T_3 \leftarrow T_5^2$
  10. $T_1 \leftarrow T_1 T_4$                $X_3 \leftarrow T_3 - T_6$
  11. $aZ_3^4 \leftarrow Z_3^2$               $T_2 \leftarrow T_2 - X_3$
  12. $T_3 \leftarrow T_5 T_2$
  13. $aZ_3^4 \leftarrow \left(aZ_3^4\right)^2$   $Y_3 \leftarrow T_3 - T_1$
  14. $aZ_3^4 \leftarrow a\left(aZ_3^4\right)$

(b)
Require: $P_1 = (X_1, Y_1, Z_1, aZ_1^4)$
Ensure: $2P_1 = P_3 = (X_3, Y_3, Z_3, aZ_3^4)$
  1.  $T_1 \leftarrow Y_1^2$                  $T_2 \leftarrow 2X_1$
  2.  $T_3 \leftarrow T_1^2$                  $T_2 \leftarrow 2T_2$
  3.  $T_1 \leftarrow T_2 T_1$                $T_3 \leftarrow 2T_3$
  4.  $T_2 \leftarrow X_1^2$                  $T_3 \leftarrow 2T_3$
  5.  $T_4 \leftarrow Y_1 Z_1$                $T_3 \leftarrow 2T_3$
  6.  $T_5 \leftarrow T_3\left(aZ_1^4\right)$     $T_6 \leftarrow 2T_2$
  7.  $T_2 \leftarrow T_6 + T_2$
  8.  $T_2 \leftarrow T_2 + \left(aZ_1^4\right)$
  9.  $T_6 \leftarrow T_2^2$                  $Z_3 \leftarrow 2T_4$
  10. $T_4 \leftarrow 2T_1$
  11. $X_3 \leftarrow T_6 - T_4$
  12. $T_1 \leftarrow T_1 - X_3$
  13. $T_2 \leftarrow T_2 T_1$                $aZ_3^4 \leftarrow 2T_5$
  14. $Y_3 \leftarrow T_2 - T_3$
```

4.7 Modular Multiplicative Inverter

Modular multiplicative inversion is done according to Fermat's theorem [8, 12], $a^{-1} = a^{p-2} \bmod p$, if $gcd(a, p) = 1$. Because the curves we are interested in are defined over $GF(p)$, p is prime, we can use this theorem to find the multiplicative inverses modulo p. So multiplicative inversion can be done by modular exponentiation of a by $p - 2$. Modular exponentiation can be realized by using the square and multiply algorithm given in [12].

MMI controls the execution of square and multiply algorithm. It includes a FSM with 4 states. The ASM chart of MMI is shown in Fig. 2.(a). The START-INV signal is the instruction signal from PtoA. The DONE-INV signal indicates that the scanning of the bits of $T1$ register is finished.

4.8 Montgomery Modular Multiplication Circuit

The i-th iteration of Step 2 in Algorithm 2 computes the temporary results

$$T_i = 2^{-1}(T_{i-1} + x_i \times Y + m_i \times N), \;\; i = 0, \cdots, l+1 \tag{3}$$

where $T_{-1} = 0$ [20]. The j-th digit of T_i is obtained using the recurrence relation

$$2^2 \times c1_{i,j} + 2 \times c0_{i,j} + t_{i,j} = t_{i-1,j+1} + x_i \times y_j + m_i \times n_j + 2 \times c1_{i,j-1} + c0_{i,j-1} \tag{4}$$

$i = 0, \cdots, l+1$, $j = 0, \cdots, l+1, c1_{i,-1} = 0$ and $c0_{i,-1} = 0$. In Eq. (4), $2 \times c1_{i,j} + c0_{i,j}$, $j = -1, \cdots, l$, denotes the carry chain up the adder.

To obtain a linear, pipelined modular multiplier, a systolic array shown in Fig. 3 is used. $X(0)$ denotes the least significant bit (LSB) of the register in which the input x is stored. T denotes the intermediate value register. The carry chain is stored in the $C0$ and $C1$ registers. The j-th cell behaves like cell (i, j), computing Eq.(4) at time $2i + j$ for $i = 0, \cdots, l+1$.

Total area of the systolic array is $(5l - 3)XOR + (7l - 7)AND + (4l - 5)OR$ gates and $4l$ flip-flops. The critical path is the same as the critical path of one regular cell and it is independent of the bit length of the operands. So it is $2T_{FA}(c_{in} \rightarrow c_{out}) + T_{HA}(c_{in} \rightarrow c_{out})$. More details can be found in [16].

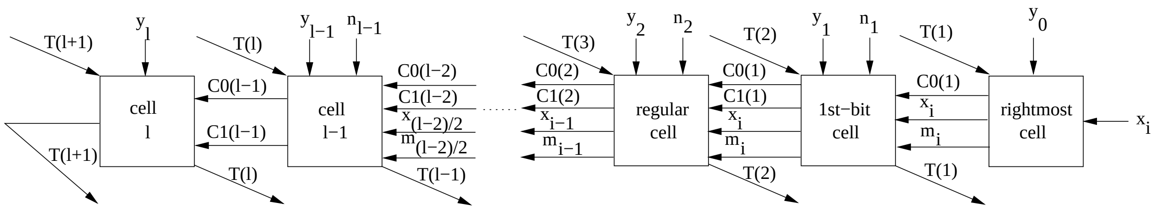

Figure 3. Schematic view of complete systolic array

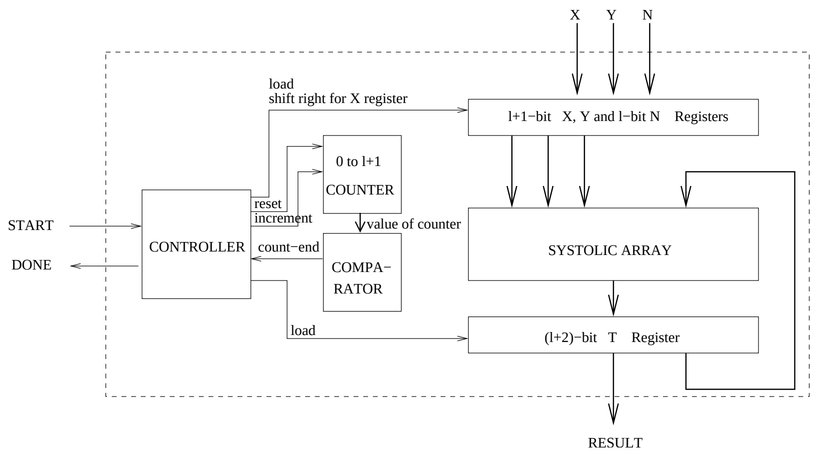

Figure 4. Architecture of the Montgomery modular multiplier circuit

The MMMC consists of a controller and a data path as shown in Fig. 4. The data path consists of a systolic array, four internal registers, a counter and a comparator.

$t_{i,j}$ is calculated at the $(2i + j)$-th clock cycle. $t_{l+1,l+1}$ is calculated at the $3l + 3$-th clock cycle. Hence, the total number of clock cycles for completing one modular Montgomery multiplication equals $3l + 3$.

4.9 Modular Addition, Substraction Circuit

Modular addition and subtraction are executed according to Algorithm 4 [4].

Algorithm 4 Modular addition and subtraction

```
Require: $M$, $0 \le A < M$, $0 \le B < M$
Ensure: $C = A + B \bmod M$
1: $C' = A + B$
2: $C'' = C' - M$
3: if $C'' < 0$ then
4:     $C = C'$
5: else
6:     $C = C''$
7: end if
```

```
Require: $M$, $0 \le A < M$, $0 \le B < M$
Ensure: $C = A - B \bmod M$
1: $C' = A - B$
2: $C'' = C' + M$
3: if $C' < 0$ then
4:     $C = C''$
5: else
6:     $C = C'$
7: end if
```

The numbers are represented in *two's complement* representation. In this representation, addition and subtraction can be realized by using the same circuit [18].

4.10 Implementation Results of The Elliptic Curve Processor

The proposed processor is implemented on Xilinx V1000E-BG-560-8 (Virtex E) FPGA by taking the bit length of EC parameters N and the bit length of k, l as 160. According to implementation results, the number of flip-flops and 4 input LUTs are 6,959 and

Table 1. Latency of the operations executed in ECP

Operation	Sub-operations	# of clock cycles depending on N and l	Execution time* ms
NtoM	4 MMM	$12N + 16$	0.021
EPM	l EC point double+ $l/2$ EC point addition	$l(51N + 66)$	14.414
PtoA	MMI+4 MMM	$3N^2 + 16N + 16$	0.397
MtoN	2 MMM	$6N + 8$	0.011
EC point doubling	8 MMM+6 MAS	$40N + 38$	0.070
EC point addition	14 MMM	$42N + 56$	0.074
MMI	$3N/2$ MMM	$9/2N^2 + 6N$	1.272
MMM		$3N + 4$	0.005
MAS		$2N + 1$	0.003

* for $N = l = 160$ at 91.308MHz

11,227, respectively. This is equivalent to 115,520 gates. Minimum clock period is 10.952ns (maximum clock frequency: 91.308MHz). LUTs are lookup-tables that are used as RAMs or 4-input gates. The latency of the operations according to the clock frequency of the implemented circuit is given in Table 1.

The only existing previous work done on FPGA is from Orlando and Paar [15]. They reported that their processor used 11,416 LUTs, 5,735 flip-flops and 35 BlockRAMs. BlockRAM is a block memory on Virtex FPGAs. On the FPGA that the authors used one BlockRAM consists of 4096 bits of memory. The clock frequency was reported as 40 MHz. If we compare both results, we can say that our processor uses less memory and can work with higher clock frequency as we expected.

5 Conclusions and Future Work

We have described an efficient implementation of a elliptic curve processor over $GF(p)$. The processor can be programmed to execute a modular multiplication, addition/subtraction, multiplicative inversion, EC point addition/doubling and multiplication. We use the method of Montgomery in a systolic array architecture for modular multiplication. Montgomery modular multiplication is proven to be very secure in hardware. Namely, the optimal bound is used which, with some savings in hardware, omits completely all reduction steps that are known to be vulnerable to side-channel attacks.

One direction in which this work should go is to implement a processor which can be programmed for point multiplication and also modular exponentiation, the basic operations for ECC and RSA, respectively. A cryptographic device dealing with both types of PKC would be very useful to secure communication systems.

Acknowledgements

Sıddıka Berna Örs is funded by a research grant of the Katholieke Universiteit Leuven, Belgium. Dr. Bart Preneel and Dr. Joos Vandewalle are professors at the Katholieke Universiteit Leuven, Belgium. This work was supported by Concerted Research Action GOA-MEFISTO-666 of the Flemish Government and by the FWO "Identification and Cryptography" project (G.0141.03).

References

[1] L. Batina, S. B. Örs, B. Preneel, and J. Vandewalle. Hardware architectures for public key cryptography. *Elsevier Science Integration the VLSI Journal*, in print, 2002.

[2] I. Blake, G. Seroussi, and N. P. Smart. *Elliptic Curves in Cryptography.* London Mathematical Society Lecture Note Series. Cambridge University Press, 1999.

[3] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast exponentiation with precomputation: Algorithms and lower bound. In R. A. Rueppel, editor, *Advances in Cryptology: Proceedings of EUROCRYPT'92*, number 658 in Lecture Notes in Computer Science, pages 200–207. Springer-Verlag, 1992.

[4] Ç. K. Koç. RSA hardware implementation. Technical report, RSA Laboratories, RSA Data Security, Inc., Redwood City, CA, August 1995.

[5] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. In K. Ohta and D. Pei, editors, *Proceedings of ASIACRYPT 1998*, number 1514 in Lecture Notes in Computer Science, pages 51–65. Springer-Verlag, 1998.

[6] J. Goodman and A. P. Chandrakasan. An energy-efficient reconfigurable public-key cryptography processor. *IEEE Journal of Solid-State Circuits*, 36(11):1808–1820, November 2001.

[7] N. Koblitz. Elliptic curve cryptosystem. *Math. Comp.*, 48:203–209, 1987.

[8] N. Koblitz. *A Course in Number Theory and Cryptography*, volume 114 of *Graduate text in mathematics.* Springer-Verlag, Berlin, Germany, second edition, 1994.

[9] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. In N. Koblitz, editor, *Advances in Cryptology: Proceedings of CRYPTO'96*, number 1109 in Lecture Notes in Computer Science, pages 104–113. Springer-Verlag, 1996.

[10] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener, editor, *Advances in Cryptology: Proceedings of CRYPTO'99*, number 1666 in Lecture Notes in Computer Science, pages 388–397. Springer-Verlag, 1999.

[11] M. M. Mano and C. R. Kime. *Logic and Computer Design Fundamentals.* Prentice Hall, Upper Saddle River, New Jersey 07458, second edition, 2001.

[12] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.

[13] V. Miller. Uses of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology: Proceedings of CRYPTO'85*, number 218 in Lecture Notes in Computer Science, pages 417–426. Springer-Verlag, 1985.

[14] P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, Vol. 44:519–521, 1985.

[15] G. Orlando and C. Paar. A scalable GF(p) elliptic curve processor architecture for programmable hardware. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Proceedings of Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2001)*, number 2162 in Lecture Notes in Computer Science, pages 356–371, Paris, France, May 14-16 2001. Springer-Verlag.

[16] S. B. Örs, L. Batina, B. Preneel, and J. Vandewalle. Hardware implementation of a Montgomery modular multiplier in a systolic array. In *The The 10th Reconfigurable Architectures Workshop (RAW)*, Nice, France, April 22 2003. to appear.

[17] H. Orup. Simplifying quotient determination in high-radix modular multiplication. In *Proceedings of the 12th Symposium on Computer Arithmetic*, pages 193–199. IEEE, 1995.

[18] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs.* Oxford University Press, Inc., New York, 2000.

[19] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[20] C. D. Walter. Montgomery's multiplication technique: How to make it smaller and faster. In Ç. K. Koç and C. Paar, editors, *Proceedings of Cryptographic Hardware and Embedded Systems (CHES 1999)*, number 1717 in Lecture Notes in Computer Science, pages 80–93. Springer-Verlag, 1999.

[21] C. D. Walter. Precise bounds for Montgomery modular multiplication and some potentially insecure RSA moduli. In B. Preneel, editor, *Proceedings of Topics in Cryptology- CT-RSA 2002*, number 2271 in Lecture Notes in Computer Science, pages 30–39, 2002.

[22] J. Wolkerstorfer. Dual-field arithmetic unit for $GF(p)$ and $GF(2^m)$. In B. S. Kaliski Jr., Ç. Koç, and C. Paar, editors, *Proceedings of Cryptographic Hardware and Embedded Systems (CHES 2002)*, Lecture Notes in Computer Science, Redwood Shores, CA, USA, August 13-15 2002. Springer-Verlag.