2016 Euromicro Conference on Digital System Design

Flexible FPGA-Based Architectures for Curve Point Multiplication over GF(p)

Dorian Amiet
IMES Institut für Mikroelektronik
und Embedded Systems
HSR Hochschule für Technik
8640 Rapperswil, Switzerland
Email: damiet@hsr.ch

Andreas Curiger
Securosys SA
8005 Zürich, Switzerland
Email: curiger@securosys.ch

Paul Zbinden
IMES Institut für Mikroelektronik
und Embedded Systems
HSR Hochschule für Technik
8640 Rapperswil, Switzerland
Email: pzbinden@hsr.ch

***Abstract*—Elliptic curve cryptography (ECC) is widely used as an efficient mechanism to secure private data using public-key protocols. This paper focuses on ECC over prime fields (GF(p)). We present a novel hardware architecture that calculates the elliptic curve point multiplication (ECPM). Our processor supports arbitrary prime fields with sizes up to 1024 bits. Different standards, which use curves in short Weierstrass form are supported. A Xilinx Virtex-7 implementation of the proposed hardware architecture takes from 0.69 ms for a 192-bit point multiplication up to 9.7 ms for 512-bit. The implementation takes only 20 DSP slices and 6816 LUTs. To the authors knowledge, this is the best performance reported so far for ECC point multiplication for arbitrary prime field curves without the use of FPGA reconfiguration.**

***Index Terms*—Elliptic curve cryptography (ECC), field programmable gate array (FPGA), finite field arithmetic, Montgomery multiplication, scalable ECC processor**

I. Introduction

Miller [1] and Koblitz [2] independently proposed elliptic curve cryptography (ECC) in 1985 as an approach to public key cryptography based on a group of points on an elliptic curve over a finite field. The security of cryptographic algorithms using the group of points on an elliptic curve is based on the elliptic curve discrete logarithm problem (ECDLP). After thirty years of research, the fastest algorithm for solving the ECDLP in this group still exhibits exponential behavior. In contrast, the DLP for other public key cryptosystems such as RSA[3] can be solved in sub-exponential time. Thus, ECC parameters can be chosen smaller than RSA for an equal level of security. Computation with smaller parameters has shown to be more efficient. Therefore ECC has recently gained more interest.

Implementations of algorithms associated with public-key cryptography in hardware accelerators are always a trade-off taken up by circuit area, throughput, and flexibility. The majority of cryptography systems using hardware accelerators relocate only computationally expensive parts to hardware accelerators. In ECC protocols, the by far most expensive part is the elliptic-curve point multiplication (ECPM).

The ECPM may be implemented either over GF(2^m) or GF(p) respectively. Many researchers have been focusing on the implementation over GF(2^m) because computation is faster than GF(p) on hardware. However, the ECC over GF(p) seems to be safer, i.e. harder to break under same key sizes. The focus of this paper is on implementations of the finite field arithmetic over GF(p).

The typical implementation for ECC applications follows a top-down structure. On top are the ECC protocols such as ECDSA (elliptic-curve digital signature algorithm) [4] or ECDH (elliptic-curve Diffie Hellman key exchange). They make use of the ECPM, which is calculated by the double-and-add algorithm (see Algorithm 2). The next layer consist of algorithms for point doubling (Algorithm PDBL, 4) and point addition (Algorithm PADD, 3). PDBL and PADD consists of modular addition, subtraction, multiplication and inversion, which form the bottom layer of operations.

Different ECPM hardware implementations have been documented for GF(p). State of the art papers in this field often describe specific applications. They are either not particularly flexible in size of parameters [5], [6] or support only a specified field [7], [8], or both [9]. In this paper, an architecture is proposed that supports different standards and user-defined curves without the necessity of reconfiguration. In other words, both, size of parameters and prime fields may be chosen arbitrarily.

The architecture presented here is based on the Montgomery multiplier from [10], the point doubling formula stems from [11] and point addition from [12]. In particular, a pipeline for the word wise digit-digit Montgomery multiplier and a parallelization of the point arithmetic is described.

This paper is organized as follows. A short mathematical background related to ECC and Montgomery multiplication is given in Section II. The hardware architecture is described in Section III. The implementation results on a Xilinx Virtex-7 FPGA and a comparison with related designs is presented in Section IV. Section V concludes this paper.

II. Background Information

In this section, a brief introduction to prime field and modular arithmetic is given.

978-1-5090-2817-7/16 $31.00 © 2016 IEEE
DOI 10.1109/DSD.2016.70

A. Elliptic Curves

An elliptic curve over a prime field GF(p) can be defined by the short Weierstrass equation (1). An elliptic curve in affine coordinates is the set of solutions, which satisfy the equation

$$y^2 = x^3 + ax + b \; (mod\, p) \tag{1}$$

where

$$p > 3 \tag{2}$$

and

$$4a^3 + 27b^2 \neq 0. \tag{3}$$

The parameters a, b and the prime p specify the curve. The variables (x, y, a, b) are all integers between 0 and $p - 1$.

Different standards for recommended curves in short Weierstrass form exist. Certicom Research standardized the secp$l$r1 [13], the German Brainpool team generated some curves named brainpoolPlr1 [14], and the most widely used curves are specified by NIST, the NIST-Pl [4]. All standards define a family of curves with different l. l defines the binary length of the parameters and lies typically between 160 and 521 in state of the art ECC algorithms. This paper will consider curves with $l \geq 192$.

B. Curve Arithmetic

Possible operations with points on a curve are PDBL and PADD. In order to calculate the ECPM based on a given point ($Q = u{\cdot}P$), l PDBL and $\approx l/2$ PADD are required. The PADD is defined by

$$\begin{aligned} R_{(x_3,y_3)} &= P_{(x_1,y_1)} + Q_{(x_2,y_2)} \\ x_3 &= \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \; (mod\, p) \\ y_3 &= \left(\frac{y_2 - y_1}{x_2 - x_1}\right) \cdot (x_1 - x_3) - y_1 \; (mod\, p) \end{aligned} \tag{4}$$

and the PDBL by

$$\begin{aligned} Q_{(x_3,y_3)} &= 2 \cdot P_{(x_1,y_1)} \\ x_3 &= \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \; (mod\, p) \\ y_3 &= \left(\frac{3x_1^2 + a}{2y_1}\right) \cdot (x_1 - x_3) - y_1 \; (mod\, p). \end{aligned} \tag{5}$$

Both operations need a computationally expensive modular inversion. An ECPM in affine coordinate requires on average $1.5 \cdot l$ inversions. This is reduced to a single inversion, if projective coordinates are used. Many different representations for projective coordinates are mentioned in the literature. This work uses Jacobian coordinates where x,y is represented as X, Y, Z, which satisfy the equation

$$\begin{aligned} x &= \frac{X}{Z^2}, \\ y &= \frac{Y}{Z^3}. \end{aligned} \tag{6}$$

Different formulas for PADD and PDBL in projective coordinates can be found in [15]. Some of them can be parallelized such that many hardware multipliers can be highly utilized. This work uses the formulas for double (Algorithm 4) from [11] and for addition (Algorithm 3) from [12].

C. Montgomery Multiplication

PADD and PDBL consist of modular addition, subtraction and multiplication. Modular reduction after an addition or subtraction is computationally trivial. However, modular reduction after multiplication is more complex. NIST and Certicom Research standardized curves, where the field p is a generalized Mersenne prime. Multiplication with modular reduction can be implemented efficiently, when the modulus is a generalized Mersenne number [4]. The Brainpool curves are not of any special form. An efficient way for modular multiplication with an arbitrary modulus is the Montgomery multiplication (MM) [16]. This operation results in

$$Z = X \cdot Y \cdot R^{-1} \; mod\, p \tag{7}$$

where $R^{-1} = 2^l$.

One can deal with the error term R^{-1} by applying the following rule: At the beginning of a calculation, all variables have to be multiplied by R: This is computed by MM by the precomputed value $R^2 \; mod\, p$.

$$XR = X \cdot R^2 \cdot R^{-1} \; mod\, p \tag{8}$$

This procedure is sometimes referred to as transformation into Montgomery space, because all following operations can be calculated with the transformed variable, if MM is used instead of modular multiplication. Modular addition and subtraction in Montgomery space are equal, since the distributive law

$$(X + Y) \cdot R = XR + YR \tag{9}$$

can be applied. After the ECPM is completed, the result has to be divided by R. The division is computed by another MM by 1.

$$Z = ZR \cdot 1 \cdot R^{-1} \; mod\, p \tag{10}$$

III. Implementation Details

This section starts with the Montgomery multiplier architecture. It shows the pipeline and the modifications of the algorithm compared to the original from [10]. Then, the parallelization of PDBL and PADD formulas are described. Furthermore, it will be shown how the final inversion can be done with least computational resources. Finally, we show how many clock cycles our architecture needs per ECPM.

A. Montgomery Multiplier

Morales-Sandoval and Diaz Perez presented in 2015 a word-based Montgomery multiplication architecture which they called iterative digit-digit Montgomery multiplication (IDDMM) [10]. Our work uses the IDDMM architecture as a starting point, because:

- Arbitrary integer sizes are supported.
- It fits perfectly into FPGAs (use of DSPs and RAMs).
- The authors reported outstanding performance in terms of area and throughput.

- No final addition is needed.

During our tests, we realized that the architecture produces false results in some cases. In particular, it turned out that the errors occurs when a carry remains at the end of the inner loop (see Algorithm 1). Thus, we expanded the IDDMM architecture by a separate mechanism for the carry. As a consequence, an additional final subtraction is needed in the algorithm. The additional parts due to the carry handling are at lines 3, 11-13, 21 and the subtraction at 24 - 26 in Algorithm 1.

Algorithm 1 Iterative digit-digit Montgomery algorithm

```
Input:
    $n = \frac{l}{k}$        ▷ $k$ = word size, $l$ = binary length of $p$
    $X = (0, X_{n-1}, ..., X_0)$        ▷ Factor 1
    $Y = (Y_{n-1}, ..., Y_0)$        ▷ Factor 2
    $p = (0, p_{n-1}, ..., p_0)$        ▷ Modulus
    $R = 2^l = \beta^n$        ▷ $\beta = 2^k$
    $p' = -p^{-1} \; mod \; \beta$
Output:
    $A = \sum_0^{n-1} A_i \beta^i = X \cdot Y \cdot R^{-1} \; mod \; p$
1: function IDDMM($X, Y$)
2:     $A_{n,...,0} \leftarrow 0$
3:     $carry \leftarrow 0$
4:     for $i \leftarrow 0$ to $n-1$ do
5:         $C \leftarrow 0$
6:         for $j \leftarrow 0$ to $n$ do
7:             $S \leftarrow A_j + X_j \cdot Y_i$
8:             if $j = 0$ then
9:                 $q \leftarrow S \cdot p' \; mod \; \beta$
10:            end if
11:            if $j = n$ then
12:                $S \leftarrow S + carry$
13:            end if
14:            $R \leftarrow q \cdot p_j$
15:            $U_j \leftarrow S + R + C \; mod \; \beta$        ▷ lower word
16:            $C \leftarrow S + R + C \; div \; \beta$        ▷ upper word
17:            if $j > 0$ then
18:                $A_{j-1} \leftarrow U_j$
19:            end if
20:        end for
21:        $carry \leftarrow C$        ▷ $carry \leftarrow 0$ or 1
22:    end for
23:    $A_n \leftarrow carry$
24:    if $A_{n,...,0} \geq p_{n-1,...,0}$ then        ▷ $A < 2 \cdot p$
25:        $A_{n-1,...,0} \leftarrow A_{n,...,0} - p_{n-1,...,0}$
26:    end if
27:    return $A_{n-1,...,0}$
28: end function
```

The computation time requires n^2 clocks, where $n = l/k$, l being the binary length of p and k the word size in bits. The value k defines the size of hardware primitives. The architecture needs two multipliers for unsigned input values of size k and two adders with input size $2k$. In this work, three versions for different k (16, 32 and 64) were implemented. Since the calculation time of a single Montgomery multiplication drops quadratically with rising k, the 16-bit version performance is significantly lower than the 32-bit version and therefore not considered any further.

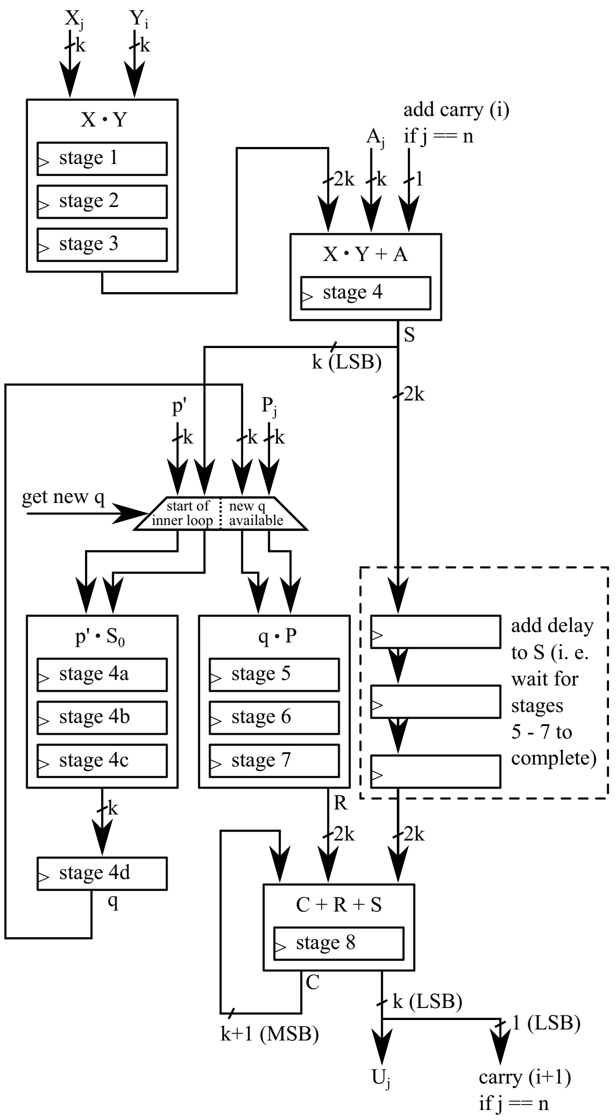


Fig. 1: The architecture of the IDDMM consists of unsigned multipliers and unsigned adders. Together with the pipelined structure, a high clock frequency can be applied and yet small latency achieved.

Our architecture needs approximately half the computation time compared to the solution from [10]. The number of clocks is given mainly by l and k. Expanding k results in higher usage of FPGA slices. To increase performance, we focused on speeding up the maximum clock of the Montgomery multiplier by reducing the length of the critical path. By adding pipeline stages to the MM architecture, the maximum clock speed could be significantly increased. As shown in Fig. 1, eight pipeline stages are created as follows: Both multiplication units contain three stages each, and the additions contain one each. However, the pipeline has a disadvantage. The input A_j, which is the outcome U_j at stage 8 from the previous iteration loop, is needed before stage 4. Filling the pipeline is only possible if $n = l/k \geq 6$. As a consequence, the version with $k = 64$ is not suited well if $l < 384$. Smaller l could be computed with the $k = 64$ version as well, but the required clock count does not decrease. Furthermore, four extra clock cycles are needed in every outer loop for the calculation of q. All in all, the proposed MM has a latency of $n^2 + 5n + 10$ clock cycles. Nevertheless, the computation time is significantly lower than the version without pipeline due to the higher clock frequency that can be applied.

The two multipliers for unsigned input values of the MM make use of the DSP slices available on Virtex-7. One slice is filled with multiplications of size $16 \cdot 16$ bit. To use larger word size, scheme (11) is being used.

$$\begin{aligned} c &= a \cdot b,\ \gamma = 2^{16} \\ a &= a_{31\ldots16} \cdot \gamma + a_{15\ldots0} \\ c_0 &= a_{15\ldots0} \cdot b_{15\ldots0} \\ c_1 &= a_{31\ldots16} \cdot b_{15\ldots0} \\ c_2 &= a_{15\ldots0} \cdot b_{31\ldots16} \\ c_3 &= a_{31\ldots16} \cdot b_{31\ldots16} \\ c &= c_3 \cdot \gamma^2 + c_2 \cdot \gamma + c_1 \cdot \gamma + c_0 \end{aligned} \tag{11}$$

Four parallelized DSP slices are spent on the $32 \cdot 32$ bit multiplication unit. Two multipliers are needed for the IDDMM architecture. Because one additional DSP is used for the final subtraction, nine DSP slices per Montgomery multiplier are used for the $k = 32$ version.

The version with $k = 64$ uses scheme (11) twice recursively. sixteen DSP slices are used in parallel per multiplication unit. Because the additions and subtractions are calculated in fabric, one MM instance occupies 32 DSP slices.

B. Point Multiplication

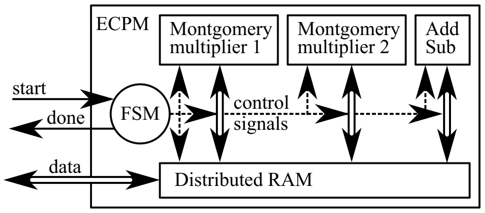


Fig. 2: The algorithms are stored in the FSM of the ECPM core

The hierarchy of the ECPM core is depicted in Fig. 2. It uses two Montgomery multipliers and one modular addition/subtraction unit. As already mentioned in the sections above, the ECPM is calculated by Algorithm 2 which uses PADD and PDBL. The complete calculation needs some additional steps.

1) Transform the starting point x_0, y_0 and the variables a and $1 (= Z_0)$ into Montgomery space.
2) Use ECPM Algorithm 2 to obtain $Q = u \cdot P_0$ in projective Jacobian coordinates.
3) Calculate the modular inversion $Z^{-1} \cdot R$ from $Z \cdot R$.
4) Compute the affine coordinates $x \cdot R,\ y \cdot R$ out of $X \cdot R,\ Y \cdot R,\ Z^{-1} \cdot R$
5) Leave Montgomery space with two final Montgomery multiplications.

These parts as well as the ECPM Algorithms (2, 3, 4) are stored in the finite state machine (FSM) of the ECPM core. Step 1, 4 and 5 need just a few Montgomery multiplications. Step 2 needs over 90% of the total computation time. This part is responsible for the decision to use two MM instances. The inversion in Step 3 is calculated using Fermat's little theorem

$$Z^{-1} = Z^{p-2} \bmod p. \tag{12}$$

The square and multiply algorithm, that might be applied for the RSA algorithm [3], can be parallelized using both Montgomery multipliers. Because both Montgomery multiplier are used, the calculation time of the inverse is equal to l times the calculation time of the Montgomery multiplication. An advantage of this method is that the computation of the inversion uses the same hardware components as the ECPM.

Algorithm 2 Double and add (MSB first)

```
Input:
    $P_0$                                        ▷ point on curve
    $u$              ▷ integer $u = u_{l-1,\ldots,0}$        ▷ $u_{l-1} = 1$
Output:
    $Q = u \cdot P_0$
 1: function PDBLADD($P_0, u$)
 2:     $Q \leftarrow P_0$
 3:     for $i \leftarrow l-2$ to $0$ do
 4:         $Q \leftarrow PDBL(Q)$                     ▷ Algorithm 4
 5:         if $u_i = 1$ then
 6:             $Q \leftarrow PADD(Q, P_0)$            ▷ Algorithm 3
 7:         end if
 8:     end for
 9:     return $Q$
10: end function
```

C. Computation Time

The double and add Algorithm 2 needs l times PDBL of Algorithm 4. The parallelization of this procedure is shown in Fig. 3a. All squares and multiplications are calculated with the IDDMM. Modular addition and subtraction is performed with one unit which takes $n + 1$ clock cycles. The PDBL computation time is around fife times the computation time of IDDMM plus some offset for additions and RAM copy. The computation time could be reduced to four times IDDMM if an additional multiplier was used. However, this would result in a lower utilization ratio of the multiplier. Utilization would fall from 90% with two multipliers down to 75% if three IDDMMs were used.

Algorithm 3 ECC point addition

```
Input:
    $X_1, Y_1, Z_1$          ▷ point, $x_1 = X_1/Z_1^2$ and $y_1 = Y_1/Z_1^3$
    $X_2, Y_2$               ▷ point in affine coordinates ($Z_2 = 1$)
Output:
    $X_3, Y_3, Z_3 = (X_1, Y_1, Z_1) + (X_2, Y_2)$
 1: function PADD($X_1, Y_1, Z_1, X_2, Y_2$)
 2:     $T_1 \leftarrow Z_1^2$                  ▷ IDDMM($Z_1, Z_1$)
 3:     $T_2 \leftarrow T_1 \cdot Z_1$
 4:     $T_1 \leftarrow T_1 \cdot X_2$
 5:     $T_2 \leftarrow T_2 \cdot Y_2$
 6:     $Z_3 \leftarrow X_1 - T_1$              ▷ modular subtraction
 7:     $T_2 \leftarrow T_2 - Y_1$
 8:     $T_4 \leftarrow Z_3^2$
 9:     $T_1 \leftarrow Z_3 \cdot T_4$
10:     $T_4 \leftarrow T_4 \cdot X_1$
11:     $X_3 \leftarrow T_2^2$
12:     $Y_3 \leftarrow T_1 \cdot Y_1$
13:     $T_3 \leftarrow 2 \cdot T_4$
14:     $X_3 \leftarrow X_3 + T_1$
15:     $X_3 \leftarrow X_3 - T_3$
16:     $T_4 \leftarrow X_3 - T_4$
17:     $T_4 \leftarrow T_4 \cdot T_2$
18:     $Z_3 \leftarrow Z_1 \cdot Z_3$
19:     $Y_3 \leftarrow T_4 - Y_3$
20:     return $X_3, Y_3, Z_3$
21: end function
```

Algorithm 4 ECC point double

```
Input:
    $X_1, Y_1, Z_1$          ▷ point, $x = X/Z^2$ and $y = Y/Z^3$
    $a$                      ▷ constant from curve (often $a = p - 3$)
Output:
    $X_3, Y_3, Z_3 = 2 \cdot (X_1, Y_1, Z_1)$
 1: function PDBL($X_1, Y_1, Z_1, a$)
 2:     $Y_3 \leftarrow Z_1^2$                  ▷ IDDMM($Z_1, Z_1$)
 3:     $X_3 \leftarrow Y_1^2$
 4:     $Z_3 \leftarrow X_1^2$
 5:     $T_1 \leftarrow a \cdot Y_3$
 6:     $T_3 \leftarrow Z_3 + Z_3$              ▷ modular addition
 7:     $T_1 \leftarrow Y_3 + T_1$
 8:     $T_2 \leftarrow X_1 \cdot X_3$
 9:     $T_3 \leftarrow T_3 + Z_3$
10:     $Y_3 \leftarrow T_3 + T_1$
11:     $T_1 \leftarrow Y_3^2$
12:     $T_3 \leftarrow T_2 + T_2$
13:     $T_2 \leftarrow X_3^2$
14:     $Z_3 \leftarrow T_3 + T_3$
15:     $T_3 \leftarrow Z_3 + Z_3$
16:     $X_3 \leftarrow T_1 - T_3$
17:     $T_1 \leftarrow Y_1 \cdot Z_1$
18:     $T_3 \leftarrow Z_3 - X_3$
19:     $T_4 \leftarrow T_3$
20:     $T_3 \leftarrow T_2 + T_2$
21:     $T_2 \leftarrow T_4 \cdot Y_3$
22:     $T_3 \leftarrow T_3 + T_3$
23:     $Y_3 \leftarrow T_3 + T_3$
24:     $Z_3 \leftarrow T_1 + T_1$
25:     $Y_3 \leftarrow T_2 - Y_3$
26:     return $X_3, Y_3, Z_3$
27: end function
```

TABLE I: Clock cycles per operation

k	l	number of clock cycles					
		+/-*	IDDMM	PDBL	PADD	inversion	point mult
32	192	7	76	440	558	15’990	155’354
	224	8	94	540	683	22’624	244’009
	256	9	114	650	820	31’232	335’360
	320	11	160	900	1130	54’400	579’440
	384	13	214	1190	1488	86’784	918’029
	512	17	346	1890	2348	185’344	1’939’558
	521	18	354	2090	2593	208’921	2’181’466
64	192	4	76	431	543	15’990	152’186
	224	5	76	434	548	18’368	196’280
	256	5	76	434	548	20’992	224’320
	320	6	76	437	553	26’240	282’352
	384	7	76	440	558	31’488	341’165
	512	9	114	650	820	62’464	670’720
	521	10	160	770	696	75’545	807’797

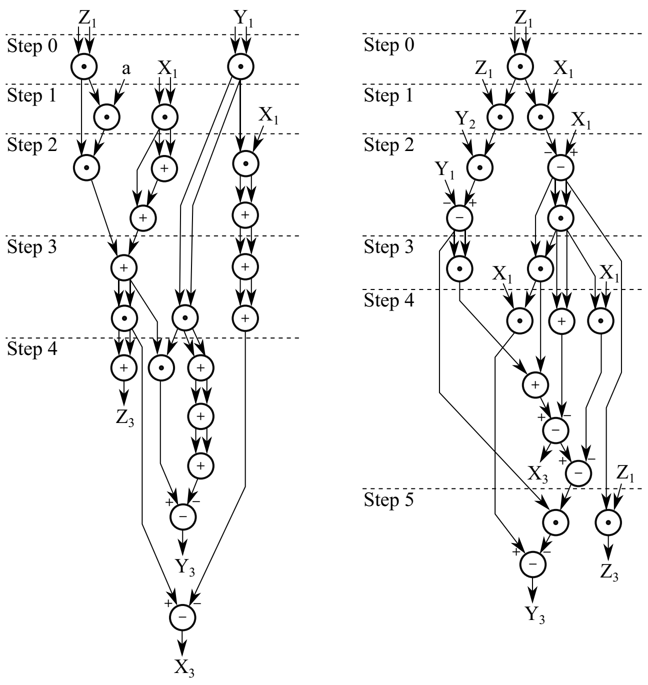
*modular addition or subtraction

Depending on u, between 0 and l PADDs are needed during the PDBLADD. Since u is a random number in most applications, this paper does the calculations based on the assumption that $l/2$ bits of u are 1 and therefore $l/2$ point additions are needed for one ECPM. Similar to the point doubling, the point addition has been parallelized using two IDDMMs (Fig. 3b). The computation time of six times IDDMM plus an offset could be reduced with an additional multiplier to five times IDDMM where the utilization ratio per IDDMM would fall from 91% to 73%.

The proposed architecture was synthesized and implemented with Vivado and tested on a Xilinx Kintex-7 FPGA (device xc7k325tffg900-2). The architecture with $k = 32$ runs with a clock speed of 219 MHz whereas the $k = 64$ version works with a 147.5 MHz clock. In Table I, the number of clocks per operation is listed. Because of the lower bound on n of the Montgomery multiplier, only curves with $l \geq 192$ are listed. Nevertheless, the architecture supports calculations on smaller curves, but the hardware efficiency is lower. However, standards using curves with $l < 192$ will become obsolete in the next few years.

D. Side Channel Considerations

So far, we have been focusing on design flexibility and performance. Let us have a closer look on security now. Obviously, Algorithms 2 is vulnerable to simple power analysis attacks (SPA, see, for instance [17]). An easy way to fix



(a) Point double, Algorithm 4 (b) Point addition, Algorithm 3

Fig. 3: This sequence is used to parallelize the computation of point double and add formulas using two Montgomery multiplier and one addition/subtraction instances.

this vulnerability is to extend Algorithm 2 with the following program stub after line 2:

```
7a:  **else then**   ⇒ $u_i = 0$
7b:      $dummy \leftarrow PADD(Q, P_0)$
7c:  **end else**
```

This extension is often referred to as double-and-add-always algorithm. The insertion of a dummy point addition renders the processing time for one point multiplication, with fixed parameter size l, constant for any u_i. Using the ECC processor extended by the above stub lets increase the latency for one point multiplication by 31%. E.g., the latency for one 256-bit point multiplication with $k = 32$ will increase from 1.49ms to 1.96ms. For a much deeper insight about side-channel analysis of our ECC processor, refer to [18].

IV. Analysis of the Results

Many hardware realizations of ECC processors have already been reported. As commented in [22], about 25% of the published processors support curves over GF(p). A fair comparison between the proposed architecture and other reported ECC processor is difficult. Older publications like [23] or [24] use old FPGA technologies that are not comparable with architecture based on new technologies. Since Vivado does not support FPGAs older than generation 7, we are not able to synthesize older technology like Virtex-5 or 6 for comparison purposes. Furthermore, some publications deal exclusively with Montgomery multiplication and not with full ECPM.

We will start this section with a comparison of Montgomery multiplication circuits, before comparing different ECPM FPGA architectures. The authors did not find any reference to an implementation with a similar degree of flexibility as the architecture proposed here. All architectures found with arbitrary l were limited to pseudo Mersenne prime fields (e. g. NIST curves) [7], [8]. Other architectures supported arbitrary fields, but had a fixed size of parameters [5] without FPGA reconfiguration. However, our ECPM implemented on a Virtex-7 (xcvu440-flgb2377-1) shows low latency and requires less FPGA resources than any other implementation.

A. *Montgomery Multiplier*

As already mentioned in section III, the proposed Montgomery multiplier architecture is based on the IDDMM from [10]. This is a very suitable implementation to compare. The multiplier has similar flexibility and functionality to our realization. In comparison to the IDDMM from [10] we reach almost double speed because of strong pipelining. On the other hand,[10] needs much less area. The main reason is that their implementation need less muxes, because:

- Our implementation uses distributed RAM for storage of the intermediate results while [10] make use of block RAM. Block RAM include all logic for parameter index muxes.
- [10] implements its IDDMM with a constant p'. This approach reduces the flexibility in the choose of the Modulus p (without runtime FPGA reconfiguration). If a p is chosen where p' becomes 1, all logic around stage 4a - 4d in Fig. 1 can be omitted.

In [19], a high performance multiplier, which calculates up to nine multiplications in parallel is inttroduced. If the pipeline is filled in an application, this multiplication has better performance in the sense of throughput per slice. However, the multiplier needs over ten times more resources and has only 3.5 times less laency compared to our 32-bit version. In addition, architecture [19] has no flexibility with respect to different operand sizes.

[20] and [21] implemented bitwise Montgomery multipliers, which are not word based. Bitwise Montgomery multipliers compute $x \cdot y_i$ in a loop during l/d cycles. Bitwise architectures fit well in LUTs, but do not make use of DSP slices. Whether the DSPs are used or not, they are still on silicon and paid for whenever FPGA is utilized. Furthermore, development and design of bit based Montgomery multipliers with flexible parameter size is difficult without using reconfiguration. However, [21] has reported low latency especially for large parameter sizes (the target application is RSA). They reach half the latency for a 1024 bit MM using eight times more LUTs in comparison to the purposed $k = 64$ version.

TABLE II: Montgomery multiplier implementation comparison

reference	arbitrary size	FPGA	area LUTs	area MULTs	max. freq. (MHz)	size(p) (bits)	latency (μs)	remarks
this work	yes	Virtex-7	1917	9	225	192 256 384 512 1024	0.338 0.507 0.951 1.54 5.31	k = 32
this work	yes	Virtex-7	2343	32	161	384 512 1024	0.472 0.708 2.149	k = 64
[10]	yes	Virtex-5	250*	11	107.9	256 512 1024	0.74 2.66 10.07	k = 32, Arch2-Ver4 *unspecified number of LUTs, estimation taken from Fig. 13a
[10]	yes	Virtex-5	704	33	68.2	256 512 1024	0.35 1.17 4.21	k = 64, Arch2-Ver4
[19]	no	Virtex-6	19362	108	197.7	256	0.147	max 9 calculations in parallel
[20]	no	Virtex-7	2361	0	152.7	256	1.68	Radix-2, $d = 1$
[21]	no	Virtex-6	10276 20527	0	222.2	512 1024	0.585 1.16	$d = 4$

TABLE III: Elliptic curve point multiplication implementation comparison

reference	arbitrary size	arbitrary curve	FPGA	area LUTs	area MULTs	frequency (MHz)	size(p) (bits)	latency (ms)	remarks
this work	yes	any	Virtex-7	6816	20	225	192 256 384 521	0.69 1.49 4.08 9.70	k = 32
this work	yes	any	Virtex-7	8273	64	161	384 521	2.12 5.02	k = 64
[7]	yes	NIST	Virtex-5	6115	7	251.3	192 256 384 521	1.71 3.95 11.81 28.04	
[8]	yes	NIST	Virtex-6	32900	289	100	192 256 384 521	0.30 0.40 1.18 1.6	
[5]	no	any	Stratix-II	6200* 9177* 12958* 17071*	92 96 177 244	160.5 157.2 150.9 145	192 256 384 512	0.44 0.68 1.35 2.23	RNS *ALM, kind of 8-input LUT
[9]	no	NIST	Virtex-4	1825* 2589*	26 32	487 490	224 256	0.45 0.62	*4-input LUT
[6]	no	any	Virtex-5	3657*	10	263	256	0.86	*slices, each with 4 LUTs
[25]	no	any	Virtex-II	1694* 1947*	2 7	108.2 68.2	no 256	29.83 15.76	*slices, each with 2 4-input LUTs

B. *Elliptic Curve Point Multiplication*

So far, we have not been able to find any ECPM architectures supporting arbitrary parameter sizes and prime fields in the literature. There are implementations with arbitrary prime fields and generic parameter size such as the one described in [5]. Other reported ECC processor implementations support flexible parameters, but they are fixed to a single curve family. E. g., [7] and [8] are specified exclusively for NIST curves. A comparison can be found in Table III.

Direct comparison between [7] and the proposed $k = 32$ version is relatively fair. [7] is disadvantaged due to the older

Virtex-5 technology, while their modular reduction scheme is fixed to the NIST pseudo Mersenne primes, which gives some advantages in area and speed. Both need roughly equivalent number of LUTs. Our architecture needs twenty instead of seven DSP slices. However, our latency is nearly three times lower.

The ECC processor from [8] is the fastest implementation presently found in the literature. It is designed as high speed ECPM and performs four times faster than our architecture for a 256 bit scalar multiplication. The price for this performance is the need for five times more LUTs and fifteen times more DSP slices.

In [5], a completely different approach to the modular reduction problem is presented. It describes an ECC processor based on the Residue Number System (RNS). Although the underlying 90nm Stratix-II FPGA means a serious disadvantage in comparison to our state of the art FPGA, the realization reaches half the latency compared to our solution. However, the RNS implementation needs much more embedded multipliers and cannot change the parameter size without FPGA reconfiguration.

In [25], a compact microcoded ECC processor has been introduced. Its small size is traded off with high processing latency. As in our design, arbitrary curves are supported as long as the parameter size is 256 bits or smaller. The utilization of microcode would suggest that changing the parameter size during runtime could theoretically be possible. With it, their design would provide a similar flexibility as our design. However, the authors did not elaborate any further.

V. Conclusion

In this paper, a highly flexible ECC processor architecture over GF(p) has been presented. The proposed design takes advantage of the high-performance DSP48E slices available on Xilinx FPGAs to increase performance. It parallelizes the PADD and PDBL operations. Modular addition, subtraction and Montgomery multiplication run in parallel. Modular inversion uses Fermat's theorem. The Virtex-7 implementation of the proposed ECC processor requires 6818 LUTs and 20 DSP48E slices. It runs at a clock frequency of 225 MHz on a Virtex-7 FPGA. It supports arbitrary curves in short Weierstrass form up to 1024 bits without the need to reconfigure the hardware. The proposed ECC processor calculates the ECPM with size 192, 224, 256, 384 and 521 in 0.69, 1.08, 1.49, 4.08 and 9.7 ms respectively. To the best of our knowledge, the proposed ECC processor is the only implementation that supports arbitrary prime field without the use of hardware reconfiguration.

Acknowledgement

This work was supported by the Commission for Technology and Innovation (CTI). The CTI is the federal innovation promotion agency responsible for encouraging science-based innovation in Switzerland.

References

[1] V. Miller, "Use of elliptic curves in cryptography," *Advances in Cryptology CRYPTO '85 Proceedings*, vol. 218, pp. 417–426, 1986.
[2] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, pp. 203–203, 1987.
[3] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, 1978.
[4] G. Locke and P. Gallagher, "FIPS PUB 186-3 Digital Signature Standard (DSS)," *Federal Information Processing Standards Publication*, vol. 3, no. June 2009, 2009.
[5] N. Guillermin, "A High Speed Coprocessor for Elliptic Curve Scalar Multiplications over Fp," *CHES 2010*, pp. 48–64, 2010.
[6] J.-Y. Lai, Y.-S. Wang, and C.-T. Huang, "High-Performance Architecture for Elliptic Curve Cryptography over Prime Fields on FPGAs," *Interdisciplinary Information Sciences*, vol. 18, no. 2, pp. 167–173, 2012.
[7] K. C. C. Loi and S.-b. Ko, "Scalable Elliptic Curve Cryptosystem FPGA Processor for NIST Prime Curves," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 11, pp. 2753–2756, 2015.
[8] H. Alrimeih and D. Rakhmatov, "Fast and Flexible Hardware Support for ECC Over Multiple Standard Prime Fields," *VLSI*, vol. 22, no. 12, pp. 2661–2674, 2014.
[9] T. Güneysu and C. Paar, "Ultra high performance ECC over NIST primes on commercial FPGAs," *Lecture Notes in Computer Science*, vol. 5154 LNCS, pp. 62–78, 2008.
[10] M. Morales-Sandoval and A. Diaz Perez, "Novel algorithms and hardware architectures for Montgomery Multiplication over GF (p)," Laboratorio de Tecnologias de la Informacion., Tech. Rep., 2015.
[11] H. Cohen, A. Miyaji, and T. Ono, "Efficient elliptic curve exponentiation using mixed coordinates," *Advances in CryptologyASIACRYPT'98*, pp. 51–65, 1998.
[12] N. T. Sullivan, "Fast Algorithms for Arithmetic on Elliptic Curves Over Prime Fields," Tech. Rep., 2008.
[13] Certicom Research, "Standards for Efficient Cryptography - SEC 2 : Recommended Elliptic Curve Domain Parameters," vol. 2, no. Sec 2, 2010.
[14] D. Harkins, "Brainpool Elliptic Curves for the Internet Key Exchange (IKE) Group Description Registry," pp. 1–12, 2013.
[15] T. L. Daniel J. Bernstein, "Explicit-formulas database, jacobian coordinates for short weierstrass curves," *http://hyperelliptic.org/*, [Online] accessed 2016-02-16.
[16] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.
[17] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," pp. 388–397, 1999.
[18] R. Willi, A. Curiger, and P. Zbinden, "On Power-Analysis Resistant Hardware Implementations of ECC-Based Cryptosystems," *Euromicro Conference on Digital System Design – DSD 2016*, 2016.
[19] X. Yan, G. Wu, D. Wu, F. Zheng, and X. Xie, "An Implementation of Montgomery Modular Multiplication on FPGAs," *2013 International Conference on Information Science and Cloud Computing*, pp. 32–38, dec 2013.
[20] M. Selim Hossain and Y. Kong, "FPGA-based efficient modular multiplication for Elliptic Curve Cryptography," *2015 International Telecommunication Networks and Applications Conference (ITNAC)*, pp. 191–195, nov 2015.
[21] G. D. Sutter, J. P. Deschamps, and J. L. 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, 2011.
[22] I. H. Hazrni, F. Zhou, F. Gebali, and T. F. Al-Somani, "Review of Elliptic Curve Processor architectures," *Communications, Computers and Signal Processing (PACRIM*, pp. 192–200, 2015.
[23] C. Mcivor, M. Mcloone, and J. Mccanny, "Hardware Elliptic Curve Cryptographic Processor Over GF(p)," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 53, no. 9, pp. 1946–1957, sep 2006.
[24] G. Orlando and C. Paar, "A Scalable GF (p) Elliptic Curve Processor Architecture for Programmable Hardware," *CHES 2001*, pp. 348–363, 2001.
[25] V. et al., "A compact FPGA-based architecture for elliptic curve cryptography over prime fields," *ASAP 2010 - 21st IEEE International Conference on Application-specific Systems, Architectures and Processors*, pp. 313–316, jul 2010.