Four$\mathbb{Q}$ on FPGA: New Hardware Speed Records for Elliptic Curve Cryptography over Large Prime Characteristic Fields

Kimmo Järvinen[1], Andrea Miele[2]*, Reza Azarderakhsh[3], and Patrick Longa[4]

[1] Aalto University, Department of Computer Science
kimmo.jarvinen@aalto.fi
[2] Intel Corporation
andrea.miele@intel.com
[3] Rochester Institute of Technology, Department of Computer Engineering
rxaeec@rit.edu
[4] Microsoft Research
plonga@microsoft.com

Abstract. We present fast and compact implementations of Four$\mathbb{Q}$ (ASIACRYPT 2015) on field-programmable gate arrays (FPGAs), and demonstrate, for the first time, the high efficiency of this new elliptic curve on reconfigurable hardware. By adapting Four$\mathbb{Q}$'s algorithms to hardware, we design FPGA-tailored architectures that are significantly faster than any other ECC alternative over large prime characteristic fields. For example, we show that our single-core and multi-core implementations can compute at a rate of 6389 and 64730 scalar multiplications per second, respectively, on a Xilinx Zynq-7020 FPGA, which represent factor-2.5 and 2 speedups in comparison with the corresponding variants of the fastest Curve25519 implementation on the same device. These results show the potential of deploying Four$\mathbb{Q}$ on hardware for high-performance and embedded security applications. All the presented implementations exhibit regular, constant-time execution, protecting against timing and simple side-channel attacks.

Keywords. Elliptic curves, Four$\mathbb{Q}$, FPGA, efficient hardware implementation, constant-time, simple side-channel attacks.

1 Introduction

With the growing deployment of elliptic curve cryptography (ECC) [15, 24] in place of traditional cryptosystems such as RSA, compact, high-performance

* This work was performed while the second author was a post-doctoral researcher at EPFL, Lausanne, Switzerland.

ECC-based implementations have become crucial for embedded systems and hardware applications. In this setting, field-programmable gate arrays (FPGAs) offer an attractive option in comparison to classical application-specific integrated circuits (ASICs), thanks to their great flexibility and faster prototyping at reduced development costs. Examples of efficient ECC implementations on FPGAs are Güneysu and Paar's implementations of the standardized NIST curves over prime fields [11] and Sasdrich and Güneysu's implementations of Curve25519 [28, 29]. There is also a plethora of FPGA implementations based on binary curves, which are particularly attractive for hardware platforms (see, e.g., [1, 2, 13, 14, 18, 26, 31]). Prime fields are by far the preferred option in software implementations mainly because efficient integer arithmetic is readily supported by instruction sets of processors. Therefore, efficient hardware implementations of ECC over large prime characteristic fields are needed to provide compatibility with software. In this work, we focus on elliptic curves defined over large prime characteristic fields.

At ASIACRYPT 2015, Costello and Longa [6] proposed a new elliptic curve called Four$\mathbb{Q}$, which provides approximately 128 bits of security and supports highly-efficient scalar multiplications by uniquely combining a four-dimensional decomposition [8] with the fastest twisted Edwards explicit formulas [12] and the efficient Mersenne prime $p = 2^{127} - 1$. In particular, by performing experiments on a large variety of software platforms, they showed that, when computing a standard variable-base scalar multiplication, Four$\mathbb{Q}$ is more than 5 times faster than the standardized NIST P-256 curve and between 2 and 3 times faster than the popular Curve25519 [5].

In this work, we propose an efficient architecture for computing scalar multiplications using Four$\mathbb{Q}$ on FPGAs. Our architecture, which leverages the power of the embedded multipliers found in modern FPGA's DSP blocks (similarly to many prior works [11, 19–23, 27–29]), supports all the necessary operations to perform Four$\mathbb{Q}$'s 4-way multi-scalar multiplication, including point validation, scalar decomposition and recoding, cofactor clearing (if required by a given protocol) and the final point conversion to affine coordinates. Based on this architecture, we designed two *high-speed* variants: a single-core architecture intended for constrained, low latency applications, and a multi-core architecture intended for high-throughput applications. Moreover, we also explore the possibility of avoiding the use of Four$\mathbb{Q}$'s endormorphisms and present an implementation variant based on the Montgomery ladder [25], which might be suitable for constrained environments. All the proposed architectures exhibit a fully regular, constant-time execution, which provides protection against timing and simple side-channel attacks (SSCA) [16, 17]. To our knowledge, these are the first implementations of Four$\mathbb{Q}$ on an FPGA in the open literature.

When compared to the most efficient FPGA implementations in the literature, our implementations show a significant increase in performance. For example, in comparison to the state-of-the-art FPGA implementation of Curve25519 by Sasdrich and Güneysu [28, 29], our single-core architecture is approximately 2.5 times faster in terms of computing time ($157\,\mu$s versus $397\,\mu$s), and our multi-core architecture is capable of computing (at full capacity) 2 times as many scalar multiplications per second as their multi-core variant (64730 scalar multiplications per second versus 32304 scalar multiplications per second). Even when comparing the case without endormorphisms, our Four$\mathbb{Q}$-based FPGA implementation is faster: the laddered variant is about 1.3 times faster than Curve25519 in terms of computing time. All these results were obtained on the same Xilinx Zynq-7020 FPGA model used by [29].

The paper is organized as follows. In Sect. 2, the relevant mathematical background and general architectural details of the proposed design are provided. In Sect. 3, the field arithmetic unit (called "the core") is presented. In Sect. 4, we describe the scalar unit consisting of the decomposition and recoding units. In Sect. 5, *three* architecture variants are detailed: single-core, multi-core and the Montgomery ladder implementation. We present the performance analysis and carry out a detailed comparison with relevant work in Sect. 6. Finally, we conclude the paper and give directions for future work in Sect. 7.

2 Preliminaries: Four$\mathbb{Q}$

Four$\mathbb{Q}$ is a high-performance elliptic curve recently proposed by Costello and Longa [6]. Given the quadratic extension field $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ with $p = 2^{127} - 1$ and $i^2 = -1$, Four$\mathbb{Q}$ is defined as the complete twisted Edwards [4] curve given by

$$\mathcal{E}/\mathbb{F}_{p^2} : \; -x^2 + y^2 = 1 + dx^2y^2, \tag{1}$$

where $d := 125317048443780598345676279555970305165 \cdot i + 4205857648805777768770$.

The set of $\mathbb{F}_{p^2}$-rational points lying on equation (1), which includes the neutral point $\mathcal{O}_{\mathcal{E}} = (0, 1)$, forms an additive abelian group. The cardinality of this group is given by $\#\mathcal{E}(\mathbb{F}_{p^2}) = 392 \cdot \xi$, where ξ is a 246-bit prime, and thus, the group $\mathcal{E}(\mathbb{F}_{p^2})[\xi]$ can be used in cryptographic systems.

The fastest set of explicit formulas for the addition law on $\mathcal{E}$ are due to Hisil, Wong, Carter and Dawson [12] using the so-called *extended twisted Edwards coordinates*: any tuple $(X : Y : Z : T)$ with $Z \neq 0$ and $T = XY/Z$ represents a projective point corresponding to an affine point $(x, y) = (X/Z, Y/Z)$. Since d is non-square over $\mathbb{F}_{p^2}$, this set of formulas is also *complete* on $\mathcal{E}$, i.e., they work without exceptions for any point in $\mathcal{E}(\mathbb{F}_{p^2})$.

Since Four$\mathbb{Q}$ is a degree-2 $\mathbb{Q}$-curve with complex multiplication [30, 10], it comes equipped with two efficiently computable endomorphisms, namely, ψ and ϕ. In [6], it is shown that these two endomorphisms enable a four-dimensional decomposition $m \mapsto (a_1, a_2, a_3, a_4) \in \mathbb{Z}^4$ for any integer $m \in [0, 2^{256} - 1]$ such that $0 \leq a_i < 2^{64}$ for $i = 1, 2, 3, 4$ (which is optimal in the context of multi-scalar multiplication) and such that a_1 is odd (which facilitates efficient, side-channel protected scalar multiplications); see [6, Proposition 5] for details about Four$\mathbb{Q}$'s decomposition procedure. This in turn induces a four-dimensional scalar multiplication with the form

$$[m]P = [a_1]P + [a_2]\phi(P) + [a_3]\psi(P) + [a_4]\phi(\psi(P)),$$

for any point $P \in \mathcal{E}(\mathbb{F}_{p^2})[\xi]$.

2.1 Scalar Multiplication Execution

Assume that the decomposition procedure in [6, Proposition 5] is applied to a given input scalar m. To execute the 4-way multi-scalar multiplication with protection against timing and SSCA attacks, one can follow [6] and use the method proposed by Faz, Longa and Sánchez [7]: the multi-scalars a_i are recoded to a representation $b_i = \sum_{i=0}^{64} b_i[j] \cdot 2^j$ with $b_i[j] \in \{-1, 0, 1\}$ for $i = 1, 2, 3, 4$, such that $b_1[j] \in \{-1, 1\}$ and $b_1[64] = 1$, and such that the recoded digits for a_2, a_3 and a_4 are "sign-aligned" with the corresponding digit from a_1, i.e., $b_i[j] \in \{0, b_1[j]\}$ for $i = 2, 3, 4$. It follows that this recoding produces exactly 65 "signed digit-columns", where a signed digit-column is defined as the value $d_j = b_1[j] + b_2[j] \cdot 2 + b_3[j] \cdot 2^2 + b_4[j] \cdot 2^3$ for $j = 0, ..., 64$. If one then precomputes the eight points $T[u] = P + u_0\phi(P) + u_1\psi(P) + u_2\phi(\psi(P))$ for $0 \leq u < 8$, where $u = (u_2, u_1, u_0)_2$, scalar multiplication—scanning the digit-columns from left to right—consists of an initial point loading and a single loop of 64 iterations, where each iteration computes one doubling and one addition with the point from $T[\,]$ corresponding to the current digit-column. Given that digit-columns are signed, one needs to negate the precomputed point before addition in the case of a negative digit-column.

Next, we recap details about the coordinate system strategy used in [6]. Costello and Longa [6] utilize *four* different point representations for $(X : Y : Z : T)$: $\mathbf{R_1} : (X, Y, Z, T_a, T_b)$, such that $T = T_a \cdot T_b$, $\mathbf{R_2} : (X+Y, Y-X, 2Z, 2dT)$, $\mathbf{R_3} : (X+Y, Y-X, Z, T)$ and $\mathbf{R_4} : (X, Y, Z)$. In the main loop of scalar multiplication, point doublings are computed as $\mathbf{R_1} \leftarrow \mathbf{R_4}$ and point additions as $\mathbf{R_1} \leftarrow \mathbf{R_1} \times \mathbf{R_2}$, where precomputed points are stored using $\mathbf{R_2}$. Note that converting point addition results from $\mathbf{R_1}$ to $\mathbf{R_4}$ (as required by inputs to point doublings) is for free: one simply ignores coordinates T_a, T_b.

2.2 High-Level Design of the Proposed Architecture

Our core design follows the same methodology described above and computes FourQ's scalar multiplication as in [6, Alg. 2]. However, there is a slight variation: since the negative of a precomputed point $(X + Y, Y - X, 2Z, 2dT)$ is given by $(Y - X, X + Y, 2Z, -2dT)$, we precompute the values $-2dT$ and store each precomputed point using the tuple $(X + Y, Y - X, 2Z, 2dT, -2dT)$. This representation is referred to as $\mathbf{R_5}$. During scalar multiplication, we simply read coordinates in the right order and assemble either $(X + Y, Y - X, 2Z, 2dT)$ (for positive digit-columns) or $(Y - X, X + Y, 2Z, -2dT)$ (for negative digit-columns). This approach completely eliminates the need for point negations during scalar multiplication at the cost of storing only 8 extra elements in $\mathbb{F}_{p^2}$. The slightly modified scalar multiplication algorithm is presented in Alg. 1.

In Alg. 2, we detail the conversion of the multi-scalars to digit-columns d_i. During a scalar multiplication, the 3-least significant bits of these digits (values "v_i") are used to select one out of eight points from the precomputed table. The top bit (values "s_i") is then used to select between the coordinate value $2dT$

Algorithm 1 Four$\mathbb{Q}$'s scalar multiplication on $\mathcal{E}(\mathbb{F}_{p^2})[\xi]$ (adapted from [6]).

```
Input: Point $P \in \mathcal{E}(\mathbb{F}_{p^2})[\xi]$ and integer scalar $m \in [0, 2^{256})$.
Output: $[m]P$.
  Compute endomorphisms:
  1: Compute $\phi(P)$, $\psi(P)$ and $\psi(\phi(P))$.
  Precompute lookup table:
  2: Compute $T[u] = P + [u_0]\phi(P) + [u_1]\psi(P) + [u_2]\psi(\phi(P))$ for $u = (u_2, u_1, u_0)_2$ in
     $0 \leq u \leq 7$. Write $T[u]$ in coordinates $(X+Y, Y-X, 2Z, 2dT, -2dT)$.
  Scalar decomposition and recoding:
  3: Decompose $m$ into the multi-scalar $(a_1, a_2, a_3, a_4)$ as in [6, Proposition 5].
  4: Recode $(a_1, a_2, a_3, a_4)$ into $(d_{64}, \ldots, d_0) = (\overline{s_{64}v_{64}}, \ldots, \overline{s_0 v_0})$ using Alg. 2. Write
     $m_i = 1$ if $s_i = 1$ and $m_i = -1$ if $s_i = 0$ for $i = 0, \ldots, 63$.
  Main loop:
  5: $Q = T[v_{64}]$
  6: for $i = 63$ to $0$ do
  7:     $Q = [2]Q$
  8:     $Q = Q + m_i \cdot T[v_i]$
  9: return $Q$
```

(if the bit is 1) and $-2dT$ (if the bit is 0), as described above for a point using representation $\mathbf{R_5}$.

The structure of Alg. 1 leads to a natural division of operations in our ECC processor. The processor consists of two main building blocks: (a) a scalar unit and (b) a field arithmetic unit. The former carries out the scalar decomposition and recoding (steps 3 and 4 in Alg. 1), and the latter—referred simply as "the core"—is responsible for computing the endomorphisms, precomputation, and the main loop through a fixed series of operations over $\mathbb{F}_{p^2}$. We describe these units in detail in Sect. 3 and Sect. 4.

3 Field Arithmetic Unit

The field arithmetic unit ("the core") performs operations in $\mathbb{F}_{p^2}$. The architecture of the core is depicted in Fig. 1. It consists of datapath (see Sect. 3.1), control logic (see Sect. 3.2), and memory. The memory is a 256×127-bit simple dual-port RAM that is implemented using BlockRAM (36Kb) resources from the FPGA device. We chose to have a 127-bit wide memory in order to minimize the overhead during memory reading and writing. This requires the use of 4 BlockRAMs which provide storage space for up to 128 $\mathbb{F}_{p^2}$ elements. As a result, storing the negative coordinate values $-2dT$ of the precomputed points as described in Sect. 2.2 comes essentially for free.

3.1 Datapath

The datapath computes operations in $\mathbb{F}_p$ and it thus operates on 127-bit operands. The datapath supports basic operations that allow the implementation of field

Algorithm 2 FourQ's multi-scalar recoding (adapted from [6]).

```
Input: Four positive integers $a_i = (0, a_i[63], \ldots, a_i[0])_2 \in \{0,1\}^{65}$ less than $2^{64}$ for
    $1 \le i \le 4$ and with $a_1$ odd.
Output: $(d_{64}, \ldots, d_0)$ with $0 \le d_i < 16$.
 1: $s_{64} = 1$
 2: for $j = 0$ to 63 do
 3:     $v_j = 0$
 4:     $s_j = a_1[j+1]$
 5:     for $i = 2$ to 4 do
 6:         $v_j = v_j + (a_i[0] \ll (i-2))$
 7:         $c = (a_1[j+1] \,|\, a_i[0]) \,\hat{}\, a_1[j+1]$
 8:         $a_i = (a_i \gg 1) + c$
 9: $v_{64} = a_2 + 2a_3 + 4a_4$
10: return $(d_{64}, \ldots, d_0) = (\overline{s_{64} v_{64}}, \ldots, \overline{s_0 v_0})$.
```

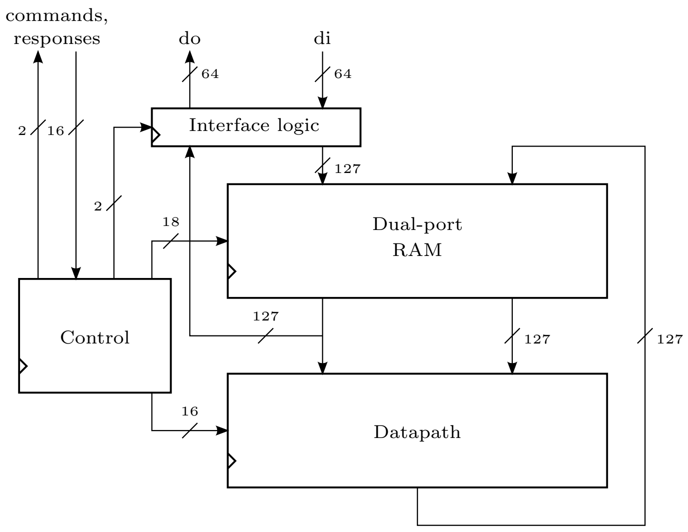

Fig. 1. Architectural diagram of the core.

multiplication, addition and subtraction. A field multiplication is performed (a) by computing a 127×127-bit integer multiplication, (b) by adding the lower and higher halves of the multiplication result to perform the first part of the reduction modulo $p = 2^{127} - 1$ and (c) by finalizing the reduction by adding the carry from the first addition. Addition and subtraction in $\mathbb{F}_p$ are computed (a) by adding/subtracting the operands and (b) by adding/subtracting the carry/borrow-bit in order to perform the modular reduction. The operations in $\mathbb{F}_{p^2}$ are implemented as a series of operations in $\mathbb{F}_p$ managed by the control logic; see Sect. 3.2. The datapath consists of two separate paths: (a) multiplier path and (b) adder/subtractor path. The datapath is shown in Fig. 2.

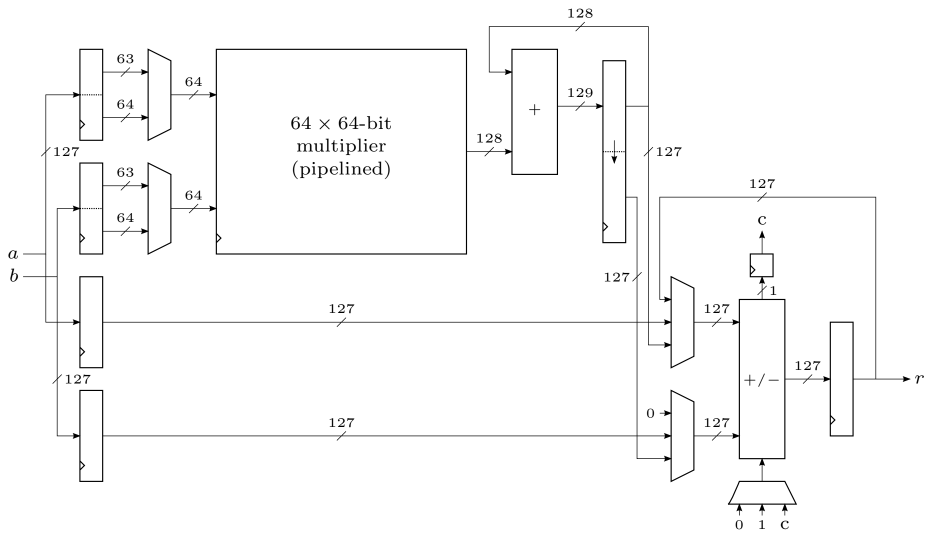

Fig. 2. The datapath for operations in $\mathbb{F}_p$.

The multiplier path is built around a pipelined 64×64-bit multiplier that is implemented using 16 hardwired multipliers (DSP blocks). The integer multiplications $a \times b$ are computed via the schoolbook algorithm. It requires four 64×64-bit partial multiplications $a_i \times b_j$ with $i, j \in \{0, 1\}$ such that $a = a_1 2^{64} + a_0$ and $b = b_1 2^{64} + b_0$. The partial multiplications are computed directly with the pipelined multiplier by selecting the operands from the input registers with two multiplexers. Results of the partial multiplications are accumulated into the upper half of a 256-bit register by using a 128-bit adder in the order $(i, j) = (0, 0), (0, 1), (1, 0), (1, 1)$. The register is shifted down by 64 bits after $(0, 0)$ and $(1, 0)$. The pipelined multiplier has *seven* pipeline stages (designed such that it matches the 128-bit adder's critical path delay).

The adder/subtractor path computes additions and subtractions as well as modular reductions over the integer multiplication results. It is built around a 127-bit adder/subtractor and multiplexers for selecting the inputs, i.e., operands and carry/borrow-bit. The value stored in the output register is the only output of the entire datapath.

The adder/subtractor path can be used for other operations while the multiplier path is performing a multiplication whenever reduction and read/write patterns of the multiplication permits it. This was achieved by including a separate set of input registers into the adder/subtractor path. In addition, the adder/subtractor path also allows accumulating the resulting value in its output register. All this allows computing most additions and subtractions required during scalar multiplication essentially for free.

3.2 Control Logic

The control logic controls the datapath and memory and, as consequence, implements all the hierarchical levels required by scalar multiplications on Four$\mathbb{Q}$. The control logic consists of a program ROM that includes instructions for the datapath and memory addresses, a small finite state machine (FSM) that controls the read addresses of the program ROM, and a recoder for recoding the instructions in the program ROM to control signals for the datapath and memory.

Field operations consist of multiple instructions that are issued by the control logic, as discussed in Sect. 3.1. Because of the pipelined multiplier, multiplications in $\mathbb{F}_p$ take several clock cycles (20 clock cycles including memory reads and writes). Fortunately, pipelining allows computing independent multiplications simultaneously and thus enables efficient operations over $\mathbb{F}_{p^2}$.

Let $a = (a_0, a_1), b = (b_0, b_1) \in \mathbb{F}_{p^2}$. Then, results (c_0, c_1) of operations in $\mathbb{F}_{p^2}$ are given by

$$\begin{aligned}
a + b &= (a_0 + b_0, a_1 + b_1) \\
a - b &= (a_0 - b_0, a_1 - b_1) \\
a \times b &= (a_0 \cdot b_0 - a_1 \cdot b_1, (a_0 + a_1) \cdot (b_0 + b_1) - a_0 \cdot b_0 - a_1 \cdot b_1) \\
a^2 &= ((a_0 + a_1) \cdot (a_0 - a_1), 2a_0 \cdot a_1) \\
a^{-1} &= (a_0 \cdot (a_0^2 + a_1^2)^{-1}, -a_1 \cdot (a_0^2 + a_1^2)^{-1})
\end{aligned}$$

where operations on the right are in $\mathbb{F}_p$. Operations in $\mathbb{F}_{p^2}$ are directly computed using the equations above: multiplication requires three field multiplications, two field additions and three field subtractions, whereas squaring requires only two field multiplications, two field additions and one field subtraction. Field inversions are computed via Fermat's Little Theorem ($a^{-1} = a^{p-2} = a^{2^{127}-3}$) using 138 multiplications in $\mathbb{F}_p$.

An example of how the control logic implements $c = a \times b$ with $a = (a_0, a_1)$ and $b = (b_0, b_1) \in \mathbb{F}_{p^2}$ using the datapath is shown in Fig. 3. The multiplication begins by computing $t_1 = a_0 \cdot b_0$ in $\mathbb{F}_p$ followed by $t_2 = a_1 \cdot b_1$. The additions $t_3 = a_0 + a_1$ and $t_4 = b_0 + b_1$ are interleaved with these multiplications. As soon as they are ready and the multiplier path becomes idle, the last multiplication $t_3 \leftarrow t_3 \cdot t_4$ is computed. The multiplication $a \times b$ ends with three successive subtractions $c_0 = t_1 - t_2$ and $c_1 = t_3 - t_1 - t_2$. The operation sequence was designed to allow the interleaving of successive multiplications over $\mathbb{F}_{p^2}$. A preceding multiplication $f = d \times e$ and subsequent multiplications $g \times h$ and $i \times j$ are depicted in gray color in Fig. 3. A multiplication finishes in 45 clock cycles but allows the next multiplication to start after only 21 clock cycles. For every other multiplication one must use t_5 in place of t_3 in order to avoid writing to t_3 before it is read. This operation sequence also allows interleaving further additions/subtractions in $\mathbb{F}_p$ with the interleaved multiplications. E.g., if we read operands from the memory in line 14, then we can compute an addition followed by a reduction in lines 16 and 17 and write the result back in line 18. There is also a variant

	Memory			Multiplier							Add/sub	
	R$_A$	R$_B$	W	Regs.	m.in	m.1	m.2$\cdots$m.5	m.6	m.7	Acc.	Regs.	Res.
1	a_0	b_0				$t_5^1 \cdot t_4^0$	$\cdots$	$d_1^1 \cdot e_1^1$		$+$		
2					t_5^1, t_4^1		$\cdots$		$d_1^1 \cdot e_1^1$	sft.		
3				a_0, b_0		$t_5^1 \cdot t_4^1$	$\cdots$	$t_5^0 \cdot t_4^0$		$+$		
4	a_0	a_1			a_0^0, b_0^0		$\cdots$		$t_5^0 \cdot t_4^0$	clr.		R-1$(d_1 \cdot e_1)$
5	b_0	b_1				$a_0^0 \cdot b_0^0$	$\cdots$	$t_5^0 \cdot t_4^1$		$+$		R-2$(d_1 \cdot e_1)$
6			t_2		a_0^0, b_0^1		$\cdots$	$t_5^1 \cdot t_4^0$	$t_5^0 \cdot t_4^1$	sft.	a_0, a_1	
7					a_0^1, b_0^0	$a_0^0 \cdot b_0^1$	$\cdots$		$t_5^1 \cdot t_4^0$	$+$	b_0, b_1	$a_0 + a_1$
8	a_1	b_1				$a_0^1 \cdot b_0^0$	$\cdots$	$t_5^1 \cdot t_4^1$		$+$		R$(a_0 + a_1)$
9			t_3		a_0^1, b_0^1		$\cdots$		$t_5^1 \cdot t_4^1$	sft.		$b_0 + b_1$
10	t_1	t_2		a_1, b_1		$a_0^1 \cdot b_0^1$	$\cdots$	$a_0^0 \cdot b_0^0$		$+$		R$(b_0 + b_1)$
11			t_4		a_1^0, b_1^0		$\cdots$		$a_0^0 \cdot b_0^0$	clr.		R-1$(t_5 \cdot t_4)$
12						$a_1^0 \cdot b_1^0$	$\cdots$	$a_0^0 \cdot b_0^1$		$+$	t_1, t_2	R-2$(t_5 \cdot t_4)$
13			t_5		a_1^0, b_1^1		$\cdots$	$a_0^1 \cdot b_0^0$	$a_0^0 \cdot b_0^1$	sft.		
14					a_1^1, b_1^0	$a_1^0 \cdot b_1^1$	$\cdots$		$a_0^1 \cdot b_0^0$	$+$		$t_1 - t_2$
15	t_3	t_4				$a_1^1 \cdot b_1^0$	$\cdots$	$a_0^1 \cdot b_0^1$		$+$		R$(t_1 - t_2)$
16			f_0		a_1^1, b_1^1		$\cdots$		$a_0^1 \cdot b_0^1$	sft.		
17	t_5	t_1		t_3, t_4		$a_1^1 \cdot b_1^1$	$\cdots$	$a_1^0 \cdot b_1^0$		$+$		
18					t_3^0, t_4^0		$\cdots$		$a_1^0 \cdot b_1^0$	clr.		R-1$(a_0 \cdot b_0)$
19		t_2				$t_3^0 \cdot t_4^0$	$\cdots$	$a_1^0 \cdot b_1^1$		$+$	t_5, t_1	R-2$(a_0 \cdot b_0)$
20			t_1		t_3^0, t_4^1		$\cdots$	$a_1^1 \cdot b_1^0$	$a_1^0 \cdot b_1^1$	sft.		$t_5 - t_1$
21					t_3^1, t_4^0	$t_3^0 \cdot t_4^1$	$\cdots$		$a_1^1 \cdot b_1^0$	$+$	t_2	R$(t_5 - t_1)$
22	g_0	h_0				$t_3^1 \cdot t_4^0$	$\cdots$	$a_1^1 \cdot b_1^1$		$+$		$R - t_2$
23					t_3^1, t_4^1		$\cdots$		$a_1^1 \cdot b_1^1$	sft.		R$(R - t_2)$
24			f_1	g_0, h_0		$t_3^1 \cdot t_4^1$	$\cdots$	$t_3^0 \cdot t_4^0$		$+$		
25	g_0	g_1			g_0^0, h_0^0		$\cdots$		$t_3^0 \cdot t_4^0$	clr.		R-1$(a_1 \cdot b_1)$
26	h_0	h_1				$g_0^0 \cdot h_0^0$	$\cdots$	$t_3^0 \cdot t_4^1$		$+$		R-2$(a_1 \cdot b_1)$
27			t_2		g_0^0, h_0^1		$\cdots$	$t_3^1 \cdot t_4^0$	$t_3^0 \cdot t_4^1$	sft.	g_0, g_1	
28					g_0^1, h_0^0	$g_0^0 \cdot h_0^1$	$\cdots$		$t_3^1 \cdot t_4^0$	$+$	h_0, h_1	$g_0 + g_1$
29	g_1	h_1				$g_0^1 \cdot h_0^0$	$\cdots$	$t_3^1 \cdot t_4^1$		$+$		R$(g_0 + g_1)$
30			t_5		g_0^1, h_0^1		$\cdots$		$t_3^1 \cdot t_4^1$	sft.		$h_0 + h_1$
31	t_1	t_2		g_1, h_1		$g_0^1 \cdot h_0^1$	$\cdots$	$g_0^0 \cdot h_0^0$		$+$		R$(h_0 + h_1)$
32			t_4		g_1^0, h_1^0		$\cdots$		$g_0^0 \cdot h_0^0$	clr.		R-1$(t_3 \cdot t_4)$
33						$g_1^0 \cdot h_1^0$	$\cdots$	$g_0^0 \cdot h_0^1$		$+$	t_1, t_2	R-2$(t_3 \cdot t_4)$
34			t_3		g_1^0, h_1^1		$\cdots$	$g_0^1 \cdot h_0^0$	$g_0^0 \cdot h_0^1$	sft.		
35					g_1^1, h_1^0	$g_1^0 \cdot h_1^1$	$\cdots$		$g_0^1 \cdot h_0^0$	$+$		$t_1 - t_2$
36	t_5	t_4				$g_1^1 \cdot h_1^0$	$\cdots$	$g_0^1 \cdot h_0^1$		$+$		R$(t_1 - t_2)$
37			c_0		g_1^1, h_1^1		$\cdots$		$g_0^1 \cdot h_0^1$	sft.		
38	t_3	t_1		t_5, t_4		$g_1^1 \cdot h_1^1$	$\cdots$	$g_1^0 \cdot h_1^0$		$+$		
39					t_5^0, t_4^0		$\cdots$		$g_1^0 \cdot h_1^0$	clr.		R-1$(g_0 \cdot h_0)$
40		t_2				$t_5^0 \cdot t_4^0$	$\cdots$	$g_1^0 \cdot h_1^1$		$+$	t_3, t_1	R-2$(g_0 \cdot h_0)$
41			t_1		t_5^0, t_4^1		$\cdots$	$g_1^1 \cdot h_1^0$	$g_1^0 \cdot h_1^1$	sft.		$t_3 - t_1$
42					t_5^1, t_4^0	$t_5^0 \cdot t_4^1$	$\cdots$		$g_1^1 \cdot h_1^0$	$+$	t_2	R$(t_3 - t_1)$
43	i_0	j_0				$t_5^1 \cdot t_4^0$	$\cdots$	$g_1^1 \cdot h_1^1$		$+$		$r - t_2$
44					t_5^1, t_4^1		$\cdots$		$g_1^1 \cdot h_1^1$	sft.		R$(r - t_2)$
45			c_1	i_0, j_0		$t_5^1 \cdot t_4^1$	$\cdots$	$t_5^0 \cdot t_4^0$		$+$		

Fig. 3. Use of the datapath for (successive) multiplications in $\mathbb{F}_{p^2}$.

of the multiplication sequence which completes the multiplication after 38 clock cycles by computing the final subtractions faster, but it does not allow efficient interleaving.

Latencies and throughputs of field operations are collected in Table 1.

The program ROM includes hand-optimized routines (fixed sequences of instructions) for all the operations required for computing scalar multiplications on FourQ. The program ROM consists of 8015 lines of instructions (13-bit ad-

Table 1. Latencies and throughputs of operations in $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$.

Operation	Latency	Throughput
Addition/subtraction in $\mathbb{F}_p$	6	1/2
Multiplication/squaring in $\mathbb{F}_p$	20	1/7
Inversion in $\mathbb{F}_p$ [†]	2760	—
Addition/subtraction in $\mathbb{F}_{p^2}$	8	1/4
Multiplication in $\mathbb{F}_{p^2}$ (max. throughput)	45	1/21
Multiplication in $\mathbb{F}_{p^2}$ (min. latency)	38	1/31
Squaring in $\mathbb{F}_{p^2}$	28	1/16
Inversion in $\mathbb{F}_{p^2}$	2817	—

[†] 126 squarings and 12 multiplications in $\mathbb{F}_p$.

dresses). Each line is 25 bits wide: 3 bits for the multiplier path, 5 bits for the adder/subtractor path, one bit for write enable and two 8-bit memory addresses for the RAM. Execution of each instruction line takes one clock cycle. We tested implementing the program ROM both using distributed memory and BlockRAM blocks. The latter resulted in slightly better timing results arguably because of an easier place-and-route process. Accordingly, we chose to implement the program ROM using 6 BlockRAM blocks.

There are in total *seven* separate routines in the program ROM. Given a basepoint $P = (x, y)$ and following Alg. 1, *initialization* (lines 1–14) assigns $X \leftarrow x$, $Y \leftarrow y$, $Z \leftarrow 1$, $T_a \leftarrow x$ and $T_b \leftarrow y$ (i.e., it maps the affine point P to representation $\mathbf{R_1}$; see Sect. 2.1). *Precomputation* (lines 15–4199) produces the table T containing 8 points using the endormorphisms and point additions. Precomputed points are stored using representation $\mathbf{R_5}$. *Initialization of the main loop* (lines 4200–4214) initializes the point accumulator by loading a point from the table T using the first digit of the recoded multi-scalar and by mapping it to representation $\mathbf{R_4}$. In the *main loop* (lines 4215–4568), point doublings $Q \leftarrow [2]Q$ and additions $Q \leftarrow Q + T[d_i]$ are computed using the representations $\mathbf{R_1} \leftarrow \mathbf{R_4}$ and $\mathbf{R_1} \leftarrow \mathbf{R_1} \times \mathbf{R_2}$, respectively. As explained in Sect. 2.1, converting precomputed points from representation $\mathbf{R_5}$ to $\mathbf{R_2}$ is simply done by reading values from memory in the right order. The main loop consists of 64 iterations and significant effort was devoted to optimizing its latency. *Affine conversion* (lines 4569–7437) maps the resulting point in representation $\mathbf{R_1}$ to affine coordinates by computing $x = X/Z$ and $y = Y/Z$. The bulk of this computation consists of an inversion in $\mathbb{F}_p$. *Point validation* (lines 7438–7561) checks if the basepoint $P = (x, y)$ is in $\mathcal{E}(\mathbb{F}_{p^2})$, i.e., it verifies that $-x^2 + y^2 - 1 - dx^2y^2 = 0$. *Cofactor clearing* (lines 7562–8014) kills the cofactor by computing $392P$. This is done with an $\mathbf{R_2} \leftarrow \mathbf{R_1}$ map (lines 7562–7643) followed by eight point doublings (lines 7644–7799) and two point additions (lines 7800–8014).

The control FSM sets the address for the program ROM depending on the phase of the scalar multiplication. The FSM includes a counter and hardcoded pointers to the routines in the program ROM. The value of the counter is used as the

Algorithm 3 Truncated multiplication algorithm.

```
Input: integers $X = X_{10}, X_9, \ldots, X_0$ in radix $2^{24}$, $Y = Y_{11}, X_{10}, \ldots, Y_0$ in radix $2^{17}$.
Output: $Z_H = \lfloor X \cdot Y / 2^{256} \rfloor \bmod 2^{64}$ or $Z_L = X \cdot Y \bmod 2^{64}$.
 1: $Z_H \leftarrow 0, Z_L \leftarrow 0$
 2: for $i = 0$ to 11 (or 3) do
 3:     $T \leftarrow 0$
 4:     for $j = 0$ to 10 (or 2) do
 5:         $T \leftarrow T + ((Y_i \cdot X_j) \ll 24j)$
 6:     $Z_H \leftarrow (Z_H \gg 17) + T$
 7:     if $i < 4$ then
 8:         $Z_L \leftarrow (Z_L \gg 17) + ((Z_H \bmod 2^{16}) \ll 51)$
 9: $Z_H \leftarrow (Z_H \gg 68) \bmod 2^{64}$
10: $Z_L \leftarrow Z_L \bmod 2^{64}$
11: return $Z_H, Z_L$
```

address to the program ROM. Depending on the operation, the FSM sets the counter to the address of the first line of the appropriate routine and, then, lets the counter count up by one every clock cycle until it reaches the end pointer of that routine. After that, the FSM jumps to the next routine or to the wait state (line 0 is no-operation).

The instruction recoder recodes instructions from the program ROM to control signals for the datapath. The memory addresses from the program ROM are fed into an address recoding circuit, which recodes the address if it is needed to access a precomputed point (otherwise, it passes the address unchanged). The address from the program ROM simply specifies the coordinate of the precomputed point and the recoding unit replaces this placeholder address with a real RAM memory address by recoding it using the value and sign of the current digit-column d_i of the scalar.

4 Scalar unit

This unit is in charge of decomposing the input scalar m into four 64-bit multi-scalars a_1, a_2, a_3, a_4, which are then recoded to a sequence of digit-columns $(d_{64}, \ldots, d_0)$ with $0 \leq d_i < 16$. These digits are used during scalar multiplication to extract the precomputed points that are to be added. In our design, this unit is naturally split into the decompose and recode units, which are described below.

4.1 Decompose unit

The decompose unit computes the multi-scalar values a_1, a_2, a_3 and a_4 as per [6, Prop. 5]. The inputs to the decompose unit are the four curve constants ℓ_1, ℓ_2, ℓ_3 and ℓ_4 and the four basis values b_1, b_2, b_3 and b_4, which are stored in a ROM, and the 256-bit input scalar m, which is stored in a register. The core

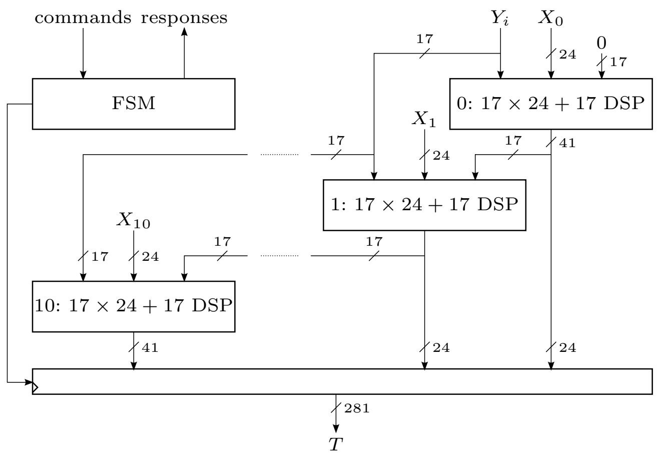

Fig. 4. Architecture of the 17x264-bit row multiplier using DSPs.

of the decompose unit is a *truncated multiplier*: on input integers $0 \leq X < 2^{256}$ and $0 \leq Y < 2^{195}$, it calculates the integer $Z_H = \lfloor X \cdot Y/(2^{256}) \rfloor \bmod 2^{64}$. This operation is needed to compute each of the four values $\widetilde{\alpha_1}$, $\widetilde{\alpha_2}$, $\widetilde{\alpha_3}$ and $\widetilde{\alpha_4}$ from [6, Prop. 5] modulo 2^{64}. The truncated multiplier computes Z_H as described in Alg. 3. In addition, this multiplier can be adapted to computations with the form $Z_L = XY \bmod 2^{64}$ by simply reducing the two for-loop counters in Alg. 3 from 11 to 3 and from 10 to 2, respectively. Thus, we reuse the truncated multiplier for the 14 multiplications modulo 2^{64} that are needed to produce the final values a_1, a_2, a_3 and a_4 as per [6, Prop. 5].

The main building block of the truncated multiplier is a 17x264-bit *row multiplier* that is used to compute the product of $Y_j \cdot X$ for some $j \in [0, 11]$ (lines 4–5 of Alg. 3). The row multiplier is implemented using a chain of 11 DSPs as shown in Fig. 4. Note that the DSP blocks available on the Xilinx Zynq FPGA family allow 17x24 unsigned integer multiplication plus addition of the result with an additional 47-bit unsigned integer. In order to comply with the operand size imposed by the DSP blocks, we split the input integer X into 24-bit words and the input Y into into 17-bit words (the most significant words are zero-padded). Both X and Y are then represented as $X_{10}, X_9, \ldots, X_0$ in radix 2^{24} and $Y_{11}, X_{10}, \ldots, Y_0$ in radix 2^{17}, respectively.

The row multiplier computes the full 17x264-bit product after 11 clock cycles. Its 281-bit result is then added to the 281-bit partial result right-shifted by 17 bits (line 6 of Alg. 3). This operation is performed by an *adder-shifter* component. In our current design, the addition has been split into 3 steps to reduce the critical path. Finally, a shift register outputs the result (line 9 of Alg. 3).

The high level architecture of the truncated multiplier unit is depicted in Fig. 5. An FSM drives the various components to execute the control statements of Alg. 3.

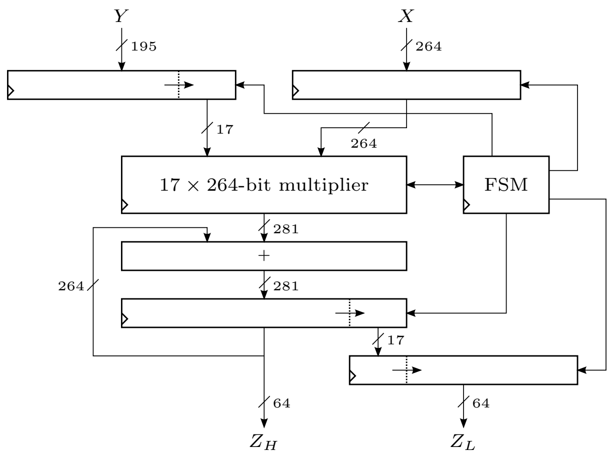

Fig. 5. Architecture of the truncated multiplier.

The remaining part of the decompose unit is an FSM that first drives the truncated multiplier to compute the four values $\widetilde{\alpha_1}$, $\widetilde{\alpha_2}$, $\widetilde{\alpha_3}$ and $\widetilde{\alpha_4}$ in four separate runnings, using as inputs the constants stored in ROM and the scalar m. For these computations, the multiplier produces outputs Z_H running for the maximum number of loop iterations according to Alg. 3. Subsequently, the FSM drives the truncated multiplier to compute products modulo 2^{64} (by running it for a reduced number of loop iterations, as explained above) and to accumulate the results Z_L to produce the output values a_1, a_2, a_3 and a_4 in 24 steps.

4.2 Recode unit

The recode unit is very simple, as the operations it performs are just bit manipulations and 64-bit additions. The unit is designed as an FSM performing 64 iterations according to Alg. 2, where each iteration is split into 6 steps (corresponding to 6 states of the FSM). The first 4 states implement lines 3 to 8 of Alg. 2, whereas the last 2 states implement line 9.

5 Architectures

We designed *three* variants of our architecture in order to provide a full picture of its capabilities compared to other designs presented in the literature.

5.1 Single-Core Architecture

Our single-core architecture is the simplest possible architecture for Alg. 1. It combines one instance of the scalar unit with one instance of the core. Most ECC hardware architectures in the literature are single-core architectures.

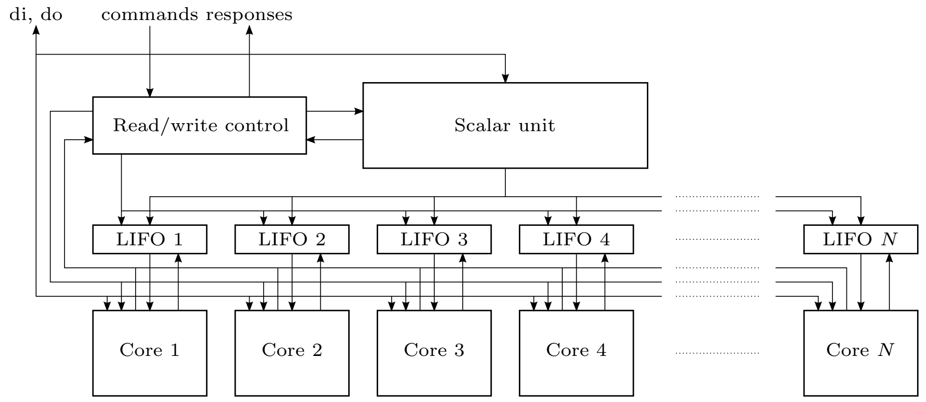

Fig. 6. The multi-core architecture with one scalar unit and N cores.

The interface of the single-core architecture is such that the host connects to the architecture through a 64-bit interface (this can be easily modified) by writing and reading values to and from the RAM. The host can issue three instructions: point validation, cofactor clearing, and scalar multiplication. Point validation computes the field operations required for computing $-x^2 + y^2 - 1 - dx^2y^2$ and the host reads the result and checks if it is zero. The need for cofactor clearing depends on the protocol and, hence, it is not included in the main scalar multiplication instruction. The scalar multiplication instruction initiates (a) the scalar unit to decompose and recode the scalar and (b) the core to begin the precomputation and all the other subsequent routines. The scalar unit computes its operations at the same time that the core computes the precomputation. Hence, scalar decomposition and recoding do not incur in any latency overhead. Once an instruction is issued, the architecture raises a busy signal which remains high as long as the operation is in process.

5.2 Multi-Core Architecture

Our multi-core architecture aims at improving throughput (operations per second). It includes one scalar unit and N instances of the core. The multi-core architecture is shown in Fig. 6. It is conceptually similar to the multi-core architecture presented by Sasdrich and Güneysu for Curve25519 in [29]. In their case, multiple cores share a common inverter unit (inversions modulo $2^{255} - 19$ are more expensive than inversions in $\mathbb{F}_{p^2}$), which is used *after* scalar multiplication. In our case the common resource is the scalar unit, which is used *at the beginning* of scalar multiplication and is computed *simultaneously* with it.

The multi-core architecture is designed so that it acts as a FIFO (first-in-first-out), which is straightforward to implement because all the operations have constant latencies. The architecture has a *busy* signal which is high when the scalar unit is computing or when all the cores are busy (or have results that have not been read by the host). The host can issue new instructions only when the busy signal is low. The cores are used cyclically so that whenever a scalar multiplication instruction is issued, the turn is given to the next core. There is also a *done* signal which is high when there are results which have not been read by the host. Reading is also performed cyclically so that the turn is handed to the next core only when the host acknowledges that it has read the previous results. This cyclic writing and reading operate independently of each other, and the interface allows reading and writing different cores. Thanks to the cyclic utilization of the cores, the interface is transparent to the host who does not need to take care of which core is actually performing the computations; in fact, that is not even visible to the host.

The scalar unit writes digits to a LIFO (last-in-first-out) buffer attached to each core. This way a core can proceed with a scalar multiplication independently of the scalar unit as soon as the scalar unit has finished decomposing and recoding a scalar. The scalar unit can then process other scalars while the previous cores are computing scalar multiplications. In this paper, we only consider situations in which a single scalar unit serves N cores. If $N > 14$, then the scalar unit becomes the bottleneck for throughput and, therefore, multiple scalar units could be required.

5.3 Architecture Using the Montgomery Ladder

The architectures above can be easily modified to compute scalar multiplications on Four$\mathbb{Q}$ without utilizing the endomorphisms. This option might be beneficial in some resource-constrained applications. To demonstrate this, we designed a modification of the single-core architecture. The main difference is that the scalar unit is no longer needed, which results in a significant reduction in the size of the architecture. Changes in the core are small and are strictly limited to the control logic. In particular, the program ROM reduces in size because of a shorter program and smaller address space (fewer temporary variables in use). The architecture accepts both 256-bit and 246-bit (reduced modulo ξ) scalars, and also supports cofactor clearing.

The size of the memory remains the same even though the memory requirements of the Montgomery ladder are relatively smaller than the single-core architecture using endormorphisms (which requires a precomputed point table). The reason for this is that the number of BlockRAMs is dictated by the width (in our case, 127 bits). Using smaller width would lead to a decrease in BlockRAM requirements but also to a lower performance. Because BlockRAMs are not the critical resource, we opted for keeping the current memory structure.

We derived hand-optimized routines for the scalar multiplication initialization and the double-and-add step using the formulas from [25]. The accumulator is initialized with $Q = (X : Z) = (1 : 0)$. One double-and-add step of the Montgomery ladder takes 228 clock cycles. Because we have an either 256-bit or 246-bit scalar, a scalar multiplication involves 256 or 246 double-and-add steps,

The scalar unit outputs digits in the order $d_0, d_1, \ldots, d_{64}$ and the core uses them in a reversed order (see Alg. 1).

Table 2. Summary of resource requirements in Xilinx Zynq-7020 XC7Z020CLG484-3.

Component	LUTs		Regs.		Slices		BRAMs		DSPs	
Single-core design										
Core	869	(1.6%)	1637	(1.5%)	490	(3.7%)	10	(7.1%)	16	(7.3%)
Scalar unit	3348	(6.3%)	2771	(2.6%)	1226	(9.2%)	0	(0.0%)	11	(5.0%)
Total	4217	(7.9%)	4413	(4.1%)	1691	(12.7%)	10	(7.1%)	27	(12.3%)
Multi-core design ($N = 11$)										
Core (min.)	902	(1.7%)	1616	(1.5%)	417	(3.1%)	10	(7.1%)	16	(7.3%)
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Core (max.)	1001	(1.9%)	1630	(1.5%)	511	(3.8%)	10	(7.1%)	16	(7.3%)
Scalar unit	3422	(6.4%)	3029	(2.8%)	1201	(9.0%)	0	(0.0%)	11	(5.0%)
Total	13595	(25.6%)	20924	(19.7%)	5697	(42.8%)	110	(78.6%)	187	(85.0%)
Single-core design, Montgomery ladder										
Core	1068	(2.0%)	1638	(1.5%)	522	(3.9%)	7	(5.0%)	16	(7.3%)
Total	1069	(2.0%)	1894	(1.8%)	565	(4.2%)	7	(5.0%)	16	(7.3%)

which take exactly 58368 or 56088 clock cycles, respectively. A final conversion to extract x from $(X : Z)$ takes 2855 clock cycles. The total cost of scalar multiplication (without cofactor clearing) is 61235 or 58967 cycles for 256-bit and 246-bit scalars, respectively. Cofactor clearing is computed with nine double-and-add steps followed by an extraction of x from $(X : Z)$ and takes 4932 cycles.

6 Results and Analysis

The three architectures from Sect. 5 were compiled with Xilinx Vivado 2015.4 to a Xilinx Zynq-7020 XC7Z020CLG484-3 FPGA, which is an all programmable system-on-chip for embedded systems. All the given results were obtained after place-and-route. Table 2 presents the area requirements of the designs. Table 3 collects latencies, timings and throughputs of the different operations supported by the designs.

The single-core design requires less than 13% of all the resources available in the targeted Zynq-7020 FPGA. Timing closure was successful with a clock constraint of 190 MHz (clock period of 5.25 ns). Hence, one scalar multiplication (without cofactor clearing) takes 156.52 μs, which means 6389 operations per second. Using Vivado tools, we analyzed the power consumption of the single-core with signal activity from post-synthesis functional simulations of ten scalar multiplications. The power estimate was 0.359 W (with high confidence level), and the energy required by one scalar multiplication was about 56.2 μJ.

The multi-core design was implemented by selecting the largest N that fitted in the Zynq-7020 FPGA. Since the DSP blocks are the critical resource and

Table 3. Performance characteristics of the designs in a Xilinx Zynq-7020 XC7Z020CLG484-3 FPGA, excluding interfacing with the host.

Operation	Latency	Time (μs)		Throughput (ops)	
	(clocks)	@190MHz	@175MHz	1×190MHz	11×175MHz
Initialization	14	0.07	0.08	—	—
Point validation	124	0.65	0.71	—	—
Cofactor clearing	1760	9.26	10.06	—	—
Precomputation	4185	22.03	23.91	—	—
Scalar multiplication, init.	15	0.08	0.09	—	—
Double-and-add	354	1.86	2.02	—	—
Affine conversion	2869	15.10	16.39	—	—
Mont. ladder, init. (256-bit)	12	0.06	—	—	—
Mont. ladder, init. (246-bit)	24	0.13	—	—	—
Mont. ladder, cofact. clr.	4932	25.96	—	—	—
Mont. ladder, double-and-add	228	1.20	—	—	—
Mont. ladder, x-coord.	2855	15.03	—	—	—
Scalar decomp. and recoding	1984	10.44	11.33	95766	88206
Scalar mult. (w/o cofact. clr.)[†]	29739	156.52	169.94	6389	64730
Scalar mult. (w/ cofact. clr.)[‡]	31499	165.78	179.99	6032	61113
Scalar mult. (Mont. ladder)[*]	58967	310.35	—	3222	—

[†] Init.+Prec.+Scalar mult. init.+ 64 × double-and-add + affine conv.

[‡] Init.+Cofactor clr.+Prec.+Scalar mult. init.+ 64 × double-and-add + affine conv.

[*] Mont. ladder, init. (246-bit) + 246 × Mont. ladder, double-and-add + Mont. ladder, x-coord.

there are 220 of them in the targeted FPGA, one can estimate room for up to 13 cores. However, Vivado was unable to place-and-route a multi-core design with $N = 13$. In practice, the largest number of admissible cores was $N = 11$ (85 % DSP utilization). Even in that case timing closure was successful only with a clock constraint of 175 MHz (clock period of 5.714 ns). This results in a small increase in the computing time for one scalar multiplication, which then takes 169.94 μs (without cofactor clearing). Throughput of the multi-core design is 64730 operations per second, which is more than ten times larger than the single-core's throughput. Hence, the multi-core design offers a significant improvement for high-demand applications in which throughput is critical.

The single-core design based on the Montgomery ladder is significantly smaller than the basic single-core design mainly because there is no scalar unit. The area requirements reduce to only 7.3 % of resources (DSP blocks) at the expense of an increase in the computing time of scalar multiplication, which in this case takes 310.35 μs (with a 246-bit scalar). Throughput becomes 3222 operations per second, which is about half of the single-core design with fast endomorphisms.

Table 4 compares our implementations with different FPGA-based designs for prime field ECC with approximately 128 bits of security. The large variety of implementation platforms (also from different vendors), elliptic curves and design features (e.g., inclusion of side-channel countermeasures or support for

Table 4. Comparison of FPGA-based designs of about 256-bit prime field ECC.

Ref.	Device	Curve	N	Resources	Time (μs)	T-put (ops)
[9]	Stratix-2	any 256-bit	1	9177 ALM, 96 DSP	680	1471
[11]	Virtex-4	NIST P-256	1	1715 LS, 32 DSP, 11 BRAM	495	2020
[11]	Virtex-4	NIST P-256	16	24574 LS, 512 DSP, 176 BRAM	n/a	24700
[19]	Virtex-5	NIST P-256	1	1980 LS, 7 DSP, 2 BRAM	3951	253
[20]	Virtex-5	any 256-bit	1	1725 LS, 37 DSP, 10 BRAM	376	2662
[22]	Virtex-2	any 256-bit	1	15755 LS, 256 MUL	3836	261
[23]	Virtex-2	any 256-bit	1	3529 LS, 36 MUL	2270	441
[27]	Virtex-5	NIST P-256	1	4505 LS, 16 DSP	570	1754
[29]	Zynq-7020	Curve25519	1	1029 LS, 20 DSP, 2 BRAM	397	2519
[29]	Zynq-7020	Curve25519	11	11277 LS, 220 DSP, 22 BRAM	397	32304
This work	Zynq-7020	Four$\mathbb{Q}$, Mont.	1	565 LS, 16 DSP, 7 BRAM	310	3222
This work	Zynq-7020	Four$\mathbb{Q}$, End.	1	1691 LS, 27 DSP, 10 BRAM	157	6389
This work	Zynq-7020	Four$\mathbb{Q}$, End.	11	5697 LS, 187 DSP, 110 BRAM	170	64730

multiple primes) make a fair comparison extremely difficult. Nevertheless, the table reveals that all of our designs compute scalar multiplications faster (in terms of computation time) than any other published FPGA-based designs.

The most straightforward comparison can be done against Sasdrich and Güneysu's implementations using Curve25519 [29] (cases without DPA countermeasures) because the designs use the same FPGA and share several similarities in terms of optimization goals and approach. Our single-core architecture is 2.67 times faster in latency and 2.54 times faster in computation time and throughput. In terms of DSP blocks (the critical resource), our architecture requires 27 and [29] requires 20. Therefore, our implementation has about 1.88 times better speed-area ratio than [29]. In the case of the multi-core architecture, we obtain a throughput that is 2 times larger than that from [29]. This speedup is achieved despite the fact that the maximum clock frequency dropped to 175 MHz in our case and we were unable to utilize all of the DSP blocks because the place-and-route failed; Sasdrich and Güneysu [29] reported results with 100 % utilization with no reduction in clock frequency, without providing a technical justification.

Even the variant without endomorphisms is faster than the design from [29]. In this case, the speedup comes from the use of a different architecture and a simpler arithmetic in $\mathbb{F}_{p^2}$ over a Mersenne prime; the simpler inversion alone saves more than 10000 clock cycles. Our architecture computes scalar multiplications on Four$\mathbb{Q}$ with 1.35 times faster latency compared to [29], but because of the lower clock frequency, throughput and computation time are only 1.28 times faster. These results showcase Four$\mathbb{Q}$'s great performance even when endomorphisms are not used (e.g., in some applications with very strict memory constraints).

7 Conclusions

We presented *three* FPGA designs for the recently proposed elliptic curve Four$\mathbb{Q}$. These architectures are able to compute one scalar multiplication in only 157 μs or, alternatively, with a maximum throughput of up to 64730 operations per second by applying parallel processing in a single Zynq-7020 FPGA. The designs are the fastest FPGA implementations of elliptic curve cryptography over large prime characteristic fields at the 128-bit security level. This extends the software results from [6] by showing that Four$\mathbb{Q}$ also offers significant speedups in hardware when compared to other elliptic curves with similar strength such as Curve25519 or NIST P-256.

Our designs are inherently protected against SSCA and timing attacks. Recent horizontal attacks (such as horizontal collision correlations [3]) can break SSCA-protected implementations by exploiting leakage from partial multiplications. Our designs compute these operations with a large 64-bit word size in a highly pipelined and parallel fashion. Nevertheless, resistance against these attacks, and other attacks that apply to scenarios in which an attacker can exploit traces from multiple scalar multiplications (e.g., differential power analysis), require further analysis. Future work involves the inclusion of strong countermeasures against such attacks.

Acknowledgments

Kimmo Järvinen's work was supported in part by the Intel Institute for Collaborative Research in Secure Computing.

Reza Azarderakhsh's work was supported by the National Science Foundation under award No. CNS-1464118 and and by the US Army Research Laboratory under award No. W911NF-16-1-0204-(68023-CS). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

1. Azarderakhsh, R., Reyhani-Masoleh, A.: Efficient FPGA implementations of point multiplication on binary Edwards and generalized Hessian curves using Gaussian normal basis. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 20(8), 1453–1466 (2012)
2. Azarderakhsh, R., Reyhani-Masoleh, A.: Parallel and high-speed computations of elliptic curve cryptography using hybrid-double multipliers. IEEE Transactions on Parallel and Distributed Systems 26(6), 1668–1677 (2015)
3. Bauer, A., Jaulmes, E., Prouff, E., Reinhard, J.R., Wild, J.: Horizontal collision correlation attack on elliptic curves. Cryptography and Communications 7(1), 91–119 (2015)

4. Bernstein, D.J., Birkner, P., Joye, M., Lange, T., Peters, C.: Twisted Edwards curves. In: Progress in Cryptology — AFRICACRYPT 2008. Lecture Notes in Computer Science, vol. 5023, pp. 389–405. Springer (2008)
5. Bernstein, D.J.: Curve25519: New Diffie-Hellman speed records. In: Public-Key Cryptography (PKC 2006). Lecture Notes in Computer Science, vol. 3958, pp. 207–228 (2006)
6. Costello, C., Longa, P.: Four$\mathbb{Q}$: Four-dimensional decompositions on a $\mathbb{Q}$-curve over the Mersenne prime. In: Advances in Cryptology — ASIACRYPT 2015. Lecture Notes in Computer Science, vol. 9452, pp. 214–235. Springer (2015), full version: `https://eprint.iacr.org/2015/565`
7. Faz-Hernández, A., Longa, P., Sánchez, A.H.: Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves (extended version). J. Cryptographic Engineering 5(1), 31–52 (2015)
8. Gallant, R., Lambert, J., Vanstone, S.: Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms. In: Advances in Cryptology — CRYPTO'01. Lecture Notes in Computer Science, vol. 2139, pp. 190–200. Springer (2001)
9. Guillermin, N.: A high speed coprocessor for elliptic curve scalar multiplications over $\mathbb{F}_p$. In: Cryptographic Hardware and Embedded Systems — CHES 2010. Lecture Notes in Computer Science, vol. 6225, pp. 48–64. Springer (2010)
10. Guillevic, A., Ionica, S.: Four-dimensional GLV via the Weil restriction. In: Advances in Cryptology — ASIACRYPT 2013. Lecture Notes in Computer Science, vol. 8269, pp. 79–96. Springer (2013)
11. Güneysu, T., Paar, C.: Ultra high performance ECC over NIST primes on commercial FPGAs. In: Cryptographic Hardware and Embedded Systems — CHES 2008. Lecture Notes in Computer Science, vol. 5154, pp. 62–78 (2008)
12. Hisil, H., Wong, K.K., Carter, G., Dawson, E.: Twisted Edwards curves revisited. In: Advances in Cryptology — ASIACRYPT 2008. Lecture Notes in Computer Science, vol. 5350, pp. 326–343. Springer (2008)
13. Järvinen, K., Skyttä, J.: On Parallelization of High-Speed Processors for Elliptic Curve Cryptography. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 16(9), 1162–1175 (Sep 2008)
14. Järvinen, K., Skyttä, J.: Optimized FPGA-based elliptic curve cryptography processor for high-speed applications. Integration, the VLSI Journal 44(4), 270–279 (2011)
15. Koblitz, N.: Elliptic Curve Cryptosystems. Mathematics of Computation 48, 203–209 (1987)
16. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Advances in Cryptology — CRYPTO'96. Lecture Notes in Computer Science, vol. 1109, pp. 104–113. Springer (1996)
17. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Advances in Cryptology — CRYPTO'99. Lecture Notes in Computer Science, vol. 1666, pp. 388–397. Springer (1999)
18. Loi, K.C.C., Ko, S.B.: High performance scalable elliptic curve cryptosystem processor for Koblitz curves. Microprocessors and Microsystems 37(4–5), 394–406 (2013)
19. Loi, K.C.C., Ko, S.B.: Scalable elliptic curve cryptosystem FPGA processor for NIST prime curves. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 23(11), 2753–2756 (Nov 2015)

20. Ma, Y., Liu, Z., Pan, W., Jing, J.: A high-speed elliptic curve cryptographic processor for generic curves over $GF(p)$. In: Selected Areas in Cryptography — SAC 2013. Lecture Notes in Computer Science, vol. 8282, pp. 421–437. Springer (2014)
21. McIvor, C.J., McLoone, M., McCanny, J.V.: An FPGA elliptic curve cryptographic accelerator over $GF(p)$. In: Proceedings of the Irish Signals and Systems Conference 2004. pp. 589–594 (2004)
22. McIvor, C.J., McLoone, M., McCanny, J.V.: Hardware elliptic curve cryptographic processor over $GF(p)$. IEEE Transactions on Circuits and Systems I: Regular Papers 55(9), 1946–1957 (Sep 2006)
23. Mentens, N.: Secure and Efficient Coprocessor Design for Cryptographic Applications on FPGAs. Ph.D. thesis, Katholieke Universiteit Leuven (Jul 2007)
24. Miller, V.S.: Use of Elliptic Curves in Cryptography. In: Advances in Cryptology — CRYPTO'85. Lecture Notes in Computer Science, vol. 218, pp. 417–426 (1986)
25. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation 48(177), 243–264 (1987)
26. Rebeiro, C., Sinha Roy, S., Mukhopadhyay, D.: Pushing the limits of high-speed $GF(2^m)$ elliptic curve scalar multiplication on FPGAs. In: Cryptographic Hardware and Embedded Systems — CHES 2012. pp. 494–511. Springer (2012)
27. Roy, D.B., Mukhopadhyay, D., Izumi, M., Takahashi, J.: Tile before multiplication: An efficient strategy to optimize DSP multiplier for accelerating prime field ECC for NIST curves. In: Proceedings of the 51st Annual Design Automation Conference—DAC'14. pp. 177:1–177:6. ACM (2014)
28. Sasdrich, P., Güneysu, T.: Efficient elliptic-curve cryptography using Curve25519 on reconfigurable devices. In: Reconfigurable Computing Architectures, Tools, and Applications — ARC 2014. Lecture Notes in Computer Science, vol. 8405, pp. 25–36. Springer (2014)
29. Sasdrich, P., Güneysu, T.: Implementing Curve25519 for side-channel-protected elliptic curve cryptography. ACM Transactions on Reconfigurable Technology and Systems 9(1), Art. 3 (2015)
30. Smith, B.: Families of fast elliptic curves from $\mathbb{Q}$-curves. In: Advances in Cryptology — ASIACRYPT 2013. Lecture Notes in Computer Science, vol. 8269, pp. 61–78. Springer (2013)
31. Sutter, G.D., Deschamps, J.P., Imaña, J.L.: Efficient elliptic curve point multiplication using digit-serial binary field operations. IEEE Transactions on Industrial Electronics 60(1), 217–225 (Jan 2013)