Implementing Complete Formulas on Weierstrass Curves in Hardware

Pedro Maat C. Massolino, Joost Renes, Lejla Batina[1]

Radboud University, Nijmegen, The Netherlands
{p.massolino,j.renes,lejla}@cs.ru.nl

Abstract. This work revisits the recent complete addition formulas for prime order elliptic curves of Renes, Costello and Batina in light of parallelization. We introduce the first hardware implementation of the new formulas on an FPGA based on three arithmetic units performing Montgomery multiplication. Our results are competitive with current literature and show the potential of the new complete formulas in hardware design. Furthermore, we present algorithms to compute the formulas using anywhere between two and six processors, using the minimum number of parallel field multiplications.

Keywords. Elliptic curve cryptography, FPGA, Weierstrass curves, Complete Addition Formulas

1 Introduction

The main operation in many cryptographic protocols based on elliptic curves is scalar multiplication, which is performed via repeated point addition and doubling. In early works formulas for the group operation used different sequences of instructions for addition and doubling [24,30]. This resulted in more optimized implementations, since doublings can be faster than general additions, but naïve implementations suffered from side-channel attacks [25]. Indeed, as all special cases have to be treated differently, it is not straightforward to come up with an efficient and side-channel secure implementation.

A class of elliptic curves which can avoid these problems is the family of curves proposed by Bernstein and Lange, the so-called Edwards curves [9]. Arguably, the primary reason for their popularity is their "complete" addition law, that is, a single addition law which can be used for all inputs. The benefit of having a complete addition law is obvious for both simplicity and side-channel security. Namely, having only one set of formulas that works for all inputs simplifies the task of implementers and helps to thwart some side-channel attacks, e. g. safe-error attacks [41]. After the introduction of Edwards curves, more curves models have been shown to possess complete addition laws [7,8]. Moreover, (twisted) Edwards curves are being deployed in software, for example in the library NaCl [11]. In particular, software implementations typically rely on specific curves, e. g. on the Montgomery curves Curve25519 [6] by Bernstein or Curve448 [21] proposed by Hamburg.

Moving to a hardware scenario, using the nice properties of these specific curves is not as straightforward anymore. Hardware development is costly, and industry prefers IP cores as generic solutions for all possible clients. Moreover, backwards compatibility is a serious concern, and curves defined over large prime fields in most standards (e. g. [13,16,31]) are

[1] This work was supported in part by the Technology Foundation STW (project 13499 - TYPHOON & ASPASIA), from the Dutch government.

prime order curves written in short Weierstrass form. This issue prohibits using (twisted) Edwards, (twisted) Hessian and Montgomery models, since these impose that the group order must be composite. The desire for complete addition formulas for prime order curves in short Weierstrass form was recognized and Renes, Costello and Batina [33] proved this to be realistic. They present complete addition formulas with an efficiency loss of 34%-44% (depending on the size of the field) in software when compared to the widely used incomplete formulas based on Jacobian coordinates.

As the authors mention, one can expect to have better performance in hardware, but they do not present results. In particular, when using Montgomery multiplication one can benefit from very efficient modular additions and subtractions (which appear a lot in their formulas), which changes the performance ratio derived in the original paper. Therefore, it is of interest to investigate the new complete formulas from a hardware point of view. In this paper we show that the hardware performance is competitive with the literature, building scalar multiplication on top of three parallel Montgomery multipliers. In more detail, we summarize our contributions as follows:

- we present the first hardware implementation based on the work of [28], working for every prime order curve over a prime field of up to 522 bits, and obtain competitive results;
- we present algorithms for various levels of parallelism for the new formulas. We show that the number of parallel multiplications decreases for up to six Montgomery multipliers.

All algorithms and their respective Magma [12] verification code can be found in Appendix B and C. The entire VHDL source code is provided on https://github.com/pmassolino/hw-triple-weierstrass.

Related work. There are numerous works on curve-based hardware implementations. Mostly on various FPGA platforms, making a meaningful comparison very difficult. Güneysu and Paar [19] proposed a new speed-optimized architecture that makes intensive use of the DSP blocks in an FPGA platform. Guillermin [20] introduced a prime field ECC hardware architecture and implemented it on several Altera FPGA boards. The design is based on Residue Number System (RNS), facilitating carry-free arithmetic and parallelism. Yao et al. [40] followed the idea of using RNS to design a high-speed ECC co-processor for pairings. Sakiyama et al. [35] proposed a superscalar coprocessor that could deal with three different curve-based cryptosystems, all in characteristic 2 fields. Varchola et al. [38] designed a processor-like architecture, with instruction set and decoder, on top of which they implemented ECC. This approach has the benefit of having a portion written in software, which can be easily maintained and updated, while having special optimized instructions for the elliptic curve operations. The downside of this approach is that the resource costs are higher than a fully optimized processor. As it was the case for Güneysu and Paar [19], their targets were standardized NIST prime curves P–224 and P–256. Consequently, each of their synthesized circuit would only work for one of the two primes. Pöpper et al. [32] follow the same approach as Varchola et al. [38], with some side-channel related improvements. The paper focuses on analyze of countermeasures and their effective cost. Roy et al. [34] followed the same path, but with more optimizations with respect to resources and only for curve NIST P–256. However, the number of Block RAMs necessary for the architecture is much larger than of Pöpper et al. [32] or Varchola et al. [38]. Fan et al. [17] created an architecture for special primes and curves, namely the standardized NIST P–192. The approach was to parallelize Montgomery multiplication and formulas for point addition and doubling on the curve. Vliegen et al. [39] attempted to reduce the resources with a small core aimed at 256-bit primes.

Organization. We start with preliminaries in Section 2, and briefly discuss parallelism for the complete formulas in Section 3. Finally we present our hardware implementation using three Montgomery multipliers in Section 4.

2 Preliminaries for elliptic curve cryptography

We briefly review the basics of elliptic curves. For a more elaborate study see [18,37].

Let $p \neq 2, 3$ be prime, $q = p^n$ for a positive integer n, and $\mathbb{F}_q$ be the finite field of q elements. Let $E/\mathbb{F}_q$ be an elliptic curve defined over $\mathbb{F}_q$ with specified point $\mathcal{O}$, which we call the *point at infinity*. Every such curve is isomorphic to a curve in short Weierstrass form

$$E : Y^2Z = X^3 + aXZ^2 + bZ^3 \subset \mathbb{P}^2$$

such that $a, b \in \mathbb{F}_q$ and $\mathcal{O} = (0 : 1 : 0)$. Any point not equal to $\mathcal{O}$ is called an *affine* point. It is easy to see that a point $(X : Y : Z)$ is affine if and only if $Z \neq 0$, hence setting $x = X/Z$ and $y = Y/Z$ it follows that the affine points correspond to (x, y) such that

$$y^2 = x^3 + ax + b.$$

The points on E form a group, with $\mathcal{O}$ as its identity element. Denote by $E(\mathbb{F}_q)$ the subgroup of $\mathbb{F}_q$-rational points of E. Its *order* is its order as a group, and in this document we are only concerned with the prime order case.

There are many ways to compute the group law on E, see [10]. These differ depending on the representation of the curve and the points. As mentioned in the introduction, we put emphasis on complete addition formulas for prime order elliptic curves. The work of Renes et al. [33] presents addition formulas for curves in short Weierstrass form with homogeneous coordinates. They compute the sum of two points $P = (X_1 : Y_1 : Z_1)$ and $Q = (X_2 : Y_2 : Z_2)$ as $P + Q = (X_3 : Y_3 : Z_3)$, where

$$\begin{aligned}
X_3 &= (X_1Y_2 + X_2Y_1)(Y_1Y_2 - a(X_1Z_2 + X_2Z_1) - 3bZ_1Z_2) \\
&\quad - (Y_1Z_2 + Y_2Z_1)(aX_1X_2 + 3b(X_1Z_2 + X_2Z_1) - a^2Z_1Z_2), \\
Y_3 &= (3X_1X_2 + aZ_1Z_2)(aX_1X_2 + 3b(X_1Z_2 + X_2Z_1) - a^2Z_1Z_2) \\
&\quad + (Y_1Y_2 + a(X_1Z_2 + X_2Z_1) + 3bZ_1Z_2)(Y_1Y_2 - a(X_1Z_2 + X_2Z_1) - 3bZ_1Z_2), \\
Z_3 &= (Y_1Z_2 + Y_2Z_1)(Y_1Y_2 + a(X_1Z_2 + X_2Z_1) + 3bZ_1Z_2) \\
&\quad + (X_1Y_2 + X_2Y_1)(3X_1X_2 + aZ_1Z_2).
\end{aligned}$$

This computes, without exception, $P + Q$ for any two points $P, Q \in E$, and has been shown to still be efficient (in software).

Elliptic-curve cryptography [24,30] commonly relies on the hard problem called the "elliptic curve discrete logarithm problem". This means that given two points P, Q on an elliptic curve, it is hard to find a scalar $k \in \mathbb{Z}$ such that $Q = kP$, if it exists. Therefore the main component of curve-based cryptosystems is the scalar multiplication operation $(k, P) \mapsto kP$. Since in many cases k is a secret, this operation is very sensitive to attacks. In particular many side-channel attacks [25,5] and correspondingly countermeasures [15] have been proposed. To ensure protection against simple power analysis (SPA) attacks it is important to use regular scalar multiplication algorithms, e. g. the Montgomery ladder [22] or Double-And-Add-Always [15], executing both an addition and a doubling operation per scalar bit.

3 Parallelism

An important way to increase the efficiency of the implementation is to use multiple Montgomery multipliers in parallel. In this section we give a brief explanation for our choice of three multipliers.

The addition formulas on which our scalar multiplication is built are shown in Algorithm 1 of [33]. We choose to ignore additions and subtractions since we assume to be relying on a Montgomery multiplier for which the cost of field multiplications is far higher than that of field additions. The total (multiplicative) cost in the most general case is $12\mathbf{M} + 2\mathbf{m_a} + 3\mathbf{m_{3b}}$[1]. Because our processors do not distinguish full multiplications and multiplications by constants, we consider this cost simply as $17\mathbf{M}$. The authors of [33] introduce optimizations for mixed addition and doubling, but in our case this only saves a single multiplication (and some additions). Since this does not make up for the price we would have to pay for the implementation of a second algorithm, we only examine the most general case. In Table 1 we show the interdependencies of the multiplications.

Stage	*Result*	*Multiplication*	*Dependent on*
0	ℓ_0	$X_1 \cdot X_2$	-
0	ℓ_1	$Y_1 \cdot Y_2$	-
0	ℓ_2	$Z_1 \cdot Z_2$	-
0	ℓ_3	$(X_1 + Y_1) \cdot (X_2 + Y_2)$	-
0	ℓ_4	$(X_1 + Z_1) \cdot (X_2 + Z_2)$	-
0	ℓ_5	$(Y_1 + Z_1) \cdot (Y_2 + Z_2)$	-
1	ℓ_6	$b_3 \cdot \ell_2$	ℓ_2
1	ℓ_7	$a \cdot \ell_2$	ℓ_2
1	ℓ_8	$a \cdot (\ell_4 - \ell_0 - \ell_2)$	ℓ_0, ℓ_2, ℓ_4
1	ℓ_9	$b_3 \cdot (\ell_4 - \ell_0 - \ell_2)$	ℓ_0, ℓ_2, ℓ_4
2	ℓ_{10}	$a \cdot (\ell_0 - \ell_7)$	ℓ_0, ℓ_7
2	ℓ_{11}	$(\ell_3 - \ell_0 - \ell_1) \cdot (\ell_1 - \ell_8 - \ell_6)$	$\ell_0, \ell_1, \ell_3, \ell_6, \ell_8$
2	ℓ_{13}	$(\ell_1 + \ell_8 + \ell_6) \cdot (\ell_1 - \ell_8 - \ell_6)$	ℓ_1, ℓ_6, ℓ_8
2	ℓ_{15}	$(\ell_5 - \ell_1 - \ell_2) \cdot (\ell_1 + \ell_8 + \ell_6)$	$\ell_1, \ell_2, \ell_5, \ell_6, \ell_8$
2	ℓ_{16}	$(\ell_3 - \ell_0 - \ell_1) \cdot (3\ell_0 + \ell_7)$	$\ell_0, \ell_1, \ell_3, \ell_7$
3	ℓ_{12}	$(\ell_5 - \ell_1 - \ell_2) \cdot (\ell_{10} + \ell_9)$	$\ell_1, \ell_2, \ell_5, \ell_9, \ell_{10}$
3	ℓ_{14}	$(3\ell_0 + \ell_7) \cdot (\ell_{10} + \ell_9)$	$\ell_0, \ell_7, \ell_9, \ell_{10}$

Table 1. Dependencies of multiplications inside the complete addition formulas

This allows us to write down algorithms for implementations running n processors in parallel. Denote by $\mathbf{M_n}$ resp. $\mathbf{a_n}$ the cost of doing n multiplications resp. additions (or subtractions) in parallel. In Table 2 we present the costs for $1 \le n \le 6$. We make the simple approximations

[1] We denote by $\mathbf{M}$, $\mathbf{m_a}$, $\mathbf{m_{3b}}$, $\mathbf{a}$ the cost of a general multiplication, a multiplication by curve constant a, a multiplication by curve constant $3b$, and an addition respectively.

that $\mathbf{M_n} = \mathbf{M}$ and $\mathbf{a_n} = \mathbf{a}$. We note that this ignores some practical aspects. For example a larger number of Montgomery multipliers can result in scheduling overhead, which we do not take into account. For our implementation we have chosen for $n = 3$, i. e. three Montgomery multipliers. This number of multipliers achieves a good area-time trade-off, while obtaining a favorable speed-up compared to $n = 1$. Moreover, the aforementioned practical issues (e. g. scheduling) are not as complicated to deal with as for larger n.

n	*Cost*	*Area* × *Time*	*Algorithm*
1	$17\mathbf{M} + 23\mathbf{a}$	$17\mathbf{M} + 23\mathbf{a}$	1 in [33]
2	$9\mathbf{M_2} + 12\mathbf{a_2}$	$18\mathbf{M} + 24\mathbf{a}$	Alg. 2
3	$6\mathbf{M_3} + 8\mathbf{a_3}$	$18\mathbf{M} + 24\mathbf{a}$	Alg. 3
4	$5\mathbf{M_4} + 7\mathbf{a_4}$	$20\mathbf{M} + 28\mathbf{a}$	Alg. 4
5	$4\mathbf{M_5} + 6\mathbf{a_5}$	$20\mathbf{M} + 30\mathbf{a}$	Alg. 5
6	$3\mathbf{M_6} + 6\mathbf{a_6}$	$18\mathbf{M} + 36\mathbf{a}$	Alg. 6

Table 2. Efficiency approximation of the number of Montgomery multipliers against the area used. All algorithms for $2 \leq n \leq 6$ are from this work.

4 Implementation of the formulas with three processors

In this section we introduce a novel hardware implementation, parallelizing the new formulas using three Montgomery processors. We make use of the Montgomery processors which have been proposed by Massolino et al. [28] for Microsemi® IGLOO2® FPGAs, for which the architecture is shown in Figure 1. We give a short description of the processor in Section 4.1, but for more details on its internals we refer to [28]. As a consequence of building on top of this processor, we target the same FPGA. However, it is straightforward to port it to other FPGAs, or even ASICs, which have a Montgomery multiplier with the same interface and instructions.

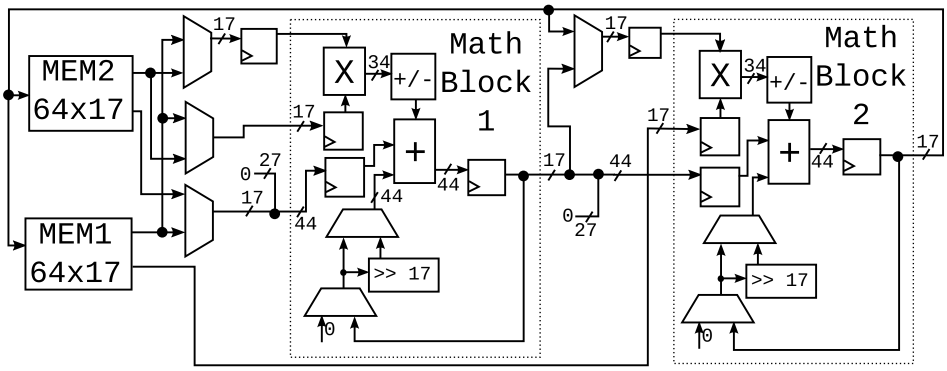

Fig. 1. The Montgomery addition, subtraction and multiplication processor.

The elliptic curve scalar multiplication routine is constructed on top of the Montgomery processors. As mentioned before, to protect against simple power analysis attacks, we implement a regular scalar multiplication algorithm (i.e. Double-And-Add-Always [15]). The algorithm relies on three registers $\mathtt{R}_0$, $\mathtt{R}_1$ and $\mathtt{R}_2$. The register $\mathtt{R}_0$ contains the operand which is always doubled. The registers $\mathtt{R}_1$ resp. $\mathtt{R}_2$ contain the result of the addition when the exponent bit is zero resp. one. This algorithm should be applied carefully since it is prone to fault attacks, see [4]. From a very high level point of view the architecture consists of the three Montgomery multipliers and a single BRAM block, shown in Figure 2. We note that this BRAM block is more than large enough to store the necessary temporary variables. So, although Algorithm 3 tries to minimize the number of these, this is not necessary for our case. In the rest of this section we elaborate on the details of the implementation.

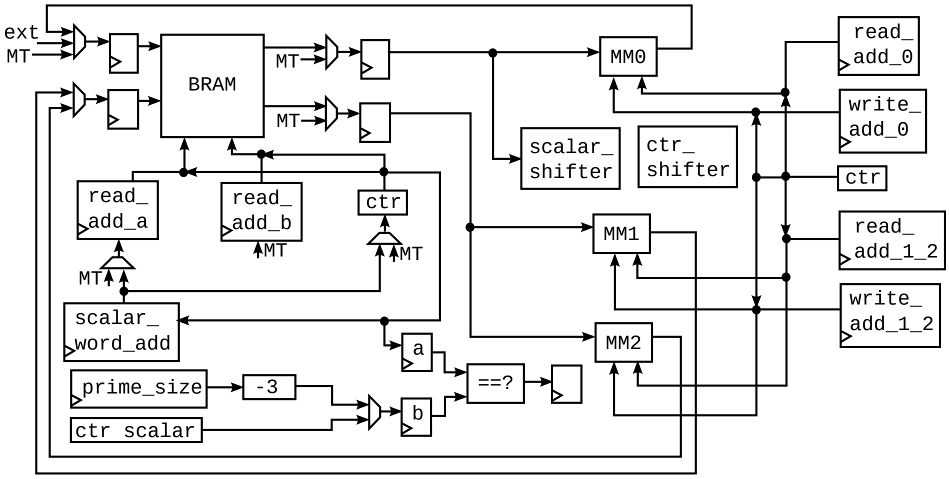

Fig. 2. The entire architecture with three Montgomery processors from [28], where `MM` = Montgomery processor, `SHR` = Shift register, `REG` = Register.

4.1 The Montgomery processor

Massolino et al. [28] proposed two different Montgomery processors. Our scalar multiplication builds on top of "version 2", which has support for two internal multipliers and two memory blocks. It can perform three operations: Montgomery multiplication, addition without reduction and subtraction without reduction. To perform Montgomery multiplication, the processor employs the FIOS algorithm proposed by Koç et al. [23]. In short, FIOS computes the partial product and partial reduction inside the same iterative loop. This can be translated into a hardware architecture, see Figure 1, with a unit for the partial product and another partial modular reduction. The circuit behaves like a three-stage pipeline: in the first stage operands are fed into the circuit, in the second they are computed and in the third they are stored into memory. The pipeline system is reused for the addition and subtraction operation in the multiplier, and values are added or subtracted directly. In case of subtraction the computation also adds a multiple of the prime modulus. Those operations can be done without applying reduction, because reduction will be applied later during a multiplication operation. However, there is a limit to the number of consecutive additions/subtractions with no reduction, on which we elaborate in Section 4.4.

4.2 Memory

The main RAM memory in Figure 2 is subdivided in order to lower control logic resources and to facilitate the interface. The main memory operates as a true dual port memory of 1024 words of 17 bits. We create a separation in the memory, composing a *big word* of 32 words (i. e. 544 bits). This way we construct the memory as 32 × 32 big words. A big word can accommodate any temporary variable, input or output of our architecture. An exception is possibly the scalar of the point scalar multiplication. Although a single word would be large enough to contain 523-bit scalars (in the largest case of a 523-bit field), the scalar blinding technique can double the size of the scalar [15]. Therefore, we use two words to store the scalar. By doing this, it will in the future be possible to execute scalar multiplication with a blinded scalar [14]. Lastly, there is a 17-bit shift register into which the scalar is loaded word by word.

4.3 Control logic

The formulas and control system are done through two state machines: a main one which controls everything, and one related to memory transfer.

The memory-transfer state machine was created with the purpose to reduce the number of states in the main machine. This was done by providing the operation of transfer between the main memory and the Montgomery processor's memory. Therefore, the main machine can transfer values with just one state, and can reuse most of the transfer logic. This memory-transfer machine becomes responsible for various parts of the bus between main memories, processors and other counters. However, the main state machine still has to be able to control everything. Hence, the main state machine shares some components with the memory transfer machine, increasing control circuit costs.

The main state machine controls all the circuits that compose the entire cryptographic core. Given that it controls the entire circuit, the machine also has the entire Table 2 scheduling implemented as states. The advantage of doing this through states is the possible optimization of the design and the entire control. However, the cost of maintenance is a lot higher than a small instruction set or microcode that can also implement the addition formulas or scalar multiplication. Because the addition formulas are complete, it is possible to reduce the costs of performing both addition and doubling through only the addition formulas. This decreases the amount of states and therefore makes the final implementation a lot more compact. Hence, the implementation only iterates over the addition formulas, until the end of the computations.

4.4 Consecutive additions

For the Montgomery processor to work in our architecture, part of the original design was changed. The authors of [28] did not need to reduce after each addition or subtraction, as they assumed that these operations would always be followed by Montgomery multiplications (and its corresponding reduction). However, they were not able to do multiple consecutive additions and subtractions, as the Montgomery division value r was chosen to be only 4 bits larger than the prime. On the other hand, it is readily seen that in Algorithm 3 there are several consecutive additions and subtractions. To be able to execute these without having to reduce, we need a Montgomery division value at least 5 bits larger than the prime. As a consequence, the processor only works for primes up to 522 bits (as opposed to 523), which is still one bit more than the largest standardized prime curve [31].

4.5 Scheduling

The architecture presented in Figure 2 has one dual port memory, whereas it has three processors. This means that we can only load values to two processors at the same time. As a consequence the three processors do not run completely in parallel, but one of the three is unsynchronized. Table 3 showcases how operations are split into different processors. They are distributed with the goal of minimizing the number of loads and stores for each processor and to minimize `MM2` being idle. The process begins by loading the necessary values into `MM0` and `MM1` and executing their respective operations. As soon as the operations in `MM0` and `MM1` are initialized, it loads the corresponding value into `MM2` and executes the operation. As soon as `MM0` and `MM1` finish their operations, this process restarts. Since the operations executed in `MM2` are not synchronized with those in `MM0` and `MM1`, both of the operations in `MM0` and `MM1` should be independent of the output of `MM2`, and vice versa. Furthermore, since multiplications are at least ten times slower than additions for our processor choice [28], the additions and subtractions from lines seven and eight in Algorithm 3 can be done by the otherwise idle processor `MM2` in stage six. This makes them basically free of cost.

Line # Alg. 3	MM0	MM1	MM2
1	$t_0 \leftarrow X_1 \cdot X_2$	$t_1 \leftarrow Y_1 \cdot Y_2$	
			$t_2 \leftarrow Z_1 \cdot Z_2$
2	$t_3 \leftarrow X_1 + Y_1$	$t_4 \leftarrow X_2 + Y_2$	
			$t_5 \leftarrow Y_1 + Z_1$
3	$t_7 \leftarrow X_1 + Z_1$	$t_8 \leftarrow X_2 + Z_2$	
			$t_6 \leftarrow Y_2 + Z_2$
4	$t_9 \leftarrow t_3 \cdot t_4$	$t_{11} \leftarrow t_7 \cdot t_8$	
			$t_{10} \leftarrow t_5 \cdot t_6$
5	$t_4 \leftarrow t_1 + t_2$	$t_5 \leftarrow t_0 + t_2$	
			$t_3 \leftarrow t_0 + t_1$
6,7,8	$t_6 \leftarrow b_3 \cdot t_2$	$t_8 \leftarrow a \cdot t_2$	
			$t_2 \leftarrow t_9 - t_3$
			$t_3 \leftarrow t_{10} - t_4$
			$t_4 \leftarrow t_{11} - t_5$
			$t_9 \leftarrow t_0 + t_0$
			$t_{10} \leftarrow t_9 + t_0$
9	$t_5 \leftarrow b_3 \cdot t_4$	$t_{11} \leftarrow a \cdot t_4$	
			$t_7 \leftarrow t_0 - t_8$
			$t_9 \leftarrow a \cdot t_7$
10	$t_0 \leftarrow t_8 + t_{10}$	$t_4 \leftarrow t_{11} + t_6$	
			$t_7 \leftarrow t_5 + t_9$
11	$t_5 \leftarrow t_1 - t_4$	$t_6 \leftarrow t_1 + t_4$	
12	$t_4 \leftarrow t_0 \cdot t_7$	$t_1 \leftarrow t_5 \cdot t_6$	
			$t_8 \leftarrow t_3 \cdot t_7$
13	$t_{11} \leftarrow t_0 \cdot t_2$	$t_9 \leftarrow t_2 \cdot t_5$	
			$t_{10} \leftarrow t_3 \cdot t_6$
14	$Y_1 \leftarrow t_1 + t_4$	$X_1 \leftarrow t_9 - t_8$	
			$Z_1 \leftarrow t_{10} + t_{11}$

Table 3. Scheduling for point addition $P \leftarrow P + Q$, where $P = (X_1 : Y_1 : Z_1)$ and $Q = (X_2 : Y_2 : Z_2)$. For doubling simply put $P = Q$.

4.6 Pre- and post-processing

For completeness, we include cycle counts for the necessary pre-processing and post-processing computations. To initialize the scalar multiplication on a point $P = (X : Y : Z)$ we perform three multiplications $X_R = X \cdot R$, $Y_R = Y \cdot R$ and $Z_R = Z \cdot R$ to put the values in the Montgomery domain. After the scalar multiplication is complete, we obtain a point $kP = (X'_R : Y'_R : Z'_R)$. Now we apply Algorithm 1 to Z'_R to obtain Z'^{-1}. Finally we compute

$x = (X'_R \cdot Z'^{-1})/R$ and $y = (Y'_R \cdot Z'^{-1})/R$, so that $kP = (x : y : 1)$. Note that all multiplications are Montgomery multiplications, and we avoid the need for an explicit division by R to exit the Montgomery domain. Table 4 presents cycle counts for pre-processing, post-processing and point addition for different field sizes.

Algorithm 1 Inversion using Fermat's little theorem

```
Require: $Z'_R$
Ensure: $Z'^{-1}$, where $Z'_R = Z' \cdot R$.
$M_0 \leftarrow 1$
$M_1 \leftarrow Z'_R$
for $0 \leq i \leq \log(p-2)$ do
    if $(p-2)_i = 1$ then
        $M_0 \leftarrow (M_1 \cdot M_0)/R$
    else
        $M_1 \leftarrow (M_1 \cdot M_1)/R$
    end if
end for
return $M_0$
```

Field size	*Pre-processing*	*Post-processing*	*Point addition*
192	570	51021	1895
224	712	75247	2311
256	870	106145	2774
320	1234	191221	3902
384	1549	278611	4874
512	2565	632915	7994
521	2565	632915	7994

Table 4. Cycle count for pre-processing, post-processing and point addition. Point doubling has the same cycle count as point addition.

4.7 Comparison

As our architecture supports primes from 116 to 522 bits, we can run benchmarks and do comparisons for multiple field sizes. The results for common prime sizes are shown in Table 6 in Appendix A. In this section we consider only the currently widely adopted 128-bit security level, presented in Table 5. Integer addition, subtraction and Montgomery modular multiplication results are the same as in Massolino et al. [28], while the cycle counts for pre-processing, post-processing and point addition are in Table 4. This is the first work implementing the new complete formulas for elliptic curves in short Weierstrass form [33] in hardware, and leads to a scalar multiplication routine which takes about 8.61ms for a 256-bit prime.

To understand our results better, we not only provide area results for IGLOO2 FPGA, but also for other technologies. This was done by describing the components architecture with behavioral VHDL. Both the behavioral VHDL and the components instantiation behave the same as in our architecture, guaranteeing correct execution. However, since they are not optimized for other FPGAs inner components, the results are only an approximation, and can likely be further improved. Nevertheless, in some cases we achieve faster scalar multiplication than in our original platform due to better and more expensive technology, e. g. in the Zynq, Virtex 5 and Virtex 6.

Even with a lot of different results, it is not straightforward to do a well-founded comparison among works in the literature. Table 5 contains different implementations of elliptic-curve scalar multiplications, but they have different optimization goals. For example we top [39] in terms of milliseconds per scalar multiplication when compared with the same FPGA and field arithmetic. On the other hand [34,26] have a trade-off between area and time, even though their proposal was aimed and optimized for a different FPGA than ours. In [38] these optimizations become more clear with a lower area and better time results on the Virtex II-Pro platform.

Other works [1,19,36,20,29,27] outperform our architecture in terms of speed, but use a much larger number of embedded multipliers. Also, implementations only focusing on NIST curves are able to use the special prime shape, yielding a significant speed-up. Depending on the needs of a specific hardware designer, this specialization of curves might not always be desirable. As mentioned before, many parties in industry might prefer generic cores. Despite these remarks, we argue that the implementation is competitive with the literature, making a similar trade-off between size and speed. Thus the new formulas can be implemented with little to no penalties, while having the benefit of not having to deal with exceptions.

5 Conclusions

In this work we provide addition formulas that can be used from one to six processors and a proof of concept architecture for three processors. The formulas can be applied not only in hardware architectures with a great array of processors, but also in software implementations that are using vector instructions. As shown in our proof of concept implementation, the formulas have competitive results with other implementations in the literature. Since our implementation is still a proof of concept, several further optimizations could be made to achieve even better results.

References

1. H. Alrimeih and D. Rakhmatov. Fast and Flexible Hardware Support for ECC Over Multiple Standard Prime Fields. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 22(12):2661–2674, Dec 2014. 10, 11, 14
2. B. Baldwin, R. R. Goundar, M. Hamilton, and W. P. Marnane. Co-Z ECC scalar multiplications for hardware, software and hardware–software co-design on embedded systems. *Journal of Cryptographic Engineering*, 2(4):221–240, 2012. 11, 14
3. B. Baldwin, R. Moloney, A. Byrne, G. McGuire, and W. P. Marnane. A Hardware Analysis of Twisted Edwards Curves for an Elliptic Curve Cryptosystem. In *Reconfigurable Computing: Architectures, Tools and Applications*, volume 5453 of *Lecture Notes in Computer Science*, pages 355–361. Springer Berlin Heidelberg, 2009. 14

Work	FPGA	Slice/	LUT	FF	Emb.	BRAM	BRAM	Freq.	Scalar Mult.	
		ALM			Mult.	64×18	1k×18	(MHz)	Cycles	(ms)
For all prime fields and prime order short Weierstrass curves										
Our	IGLOO 2^4	–	2967	1159	6	6	1	165	1421392	8.61
Our	SmartFusion 2^4	–	2775	1086	6	6	1	139	1421392	10.23
Our	Zynq$^{6\clubsuit}$	805	1911	1249	12	0	1	178	1421392	7.99
Our	Spartan 3E^4	2059	3632	1469	6	0	7	84	1421392	16.92
Our	Spartan 6^6	758	1605	1246	12	0	1	103	1421392	13.80
Our	Virtex 4^4	1414	2333	1171	6	0	1	147	1421392	9.67
Our	Virtex $5^{6\clubsuit}$	941	2231	1362	12	0	1	175	1421392	8.12
Our	Virtex $6^{6\clubsuit}$	775	1910	1250	12	0	1	152	1421392	9.35
Our	Virtex II Pro^4	1541	2562	1365	6	0	13	126	1421392	11.28
For NIST curves [31] only										
[38]	SmartFusion^4	–	3690	3690	0	0	12	109	2103941	19.3
[38]	Virtex II Pro^4	773	*1546	*1546	1	0	3	210	2103941	10.02
[38]	Virtex II Pro^4	1158	*2316	*2316	4	0	3	210	949951	4.52
[32]	Virtex $5^{6\clubsuit}$	1914	*7656	*7656	4	0	12	210	830000	3.95
[34]	Spartan 6^6	72	193	35	8	0	24	156.25	†1906250	12.20
[26]	Virtex 4^4	7020	12435	3545	8	0	4	182	†993174	5.46
[1]	Virtex $6^{6\clubsuit}$	11.2k	32.9k	*89.6k	289	0	256	100	39922	0.40
[19]	Virtex 4^4	1715	2589	2028	32	0	11	490	303450	0.62
For only Edwards or Twisted Edwards curves										
[36]	Zynq$^{6\clubsuit}$	1029	2783	3592	20	0	4	200	64770	0.32
For only specific field size, but works with any prime										
[39]	Virtex II Pro^4	1832	*3664	*3664	2	0	9	108.2	3227993	29.83
[39]	Virtex II Pro^4	2085	*4170	*4170	7	0	9	68.17	1074625	15.76
[20]	Stratix II^4	9177	*18354	*18354	96	0	0	157.2	†106896	0.68
[29]	Virtex II Pro^4	15755	*31510	*31510	256	0	0	39.46	151360	3.86
[27]	Virtex 4^4	4655	5740	4876	37	0	11	250	109297	0.44
[2]	Virtex $5^{6\clubsuit}$	2284	7822	5780	0	0	0	81.71	†331200	4.04

* Maximum possible value assumed from the number of slices. Virtex II Pro and Spartan 3E slice is 2 LUTs and FFs, Virtex 5 is 4 LUTs and FFs, finally Virtex 6 is 4 LUTs and 8 FFs. Stratix II ALM can be configured into 2 LUTs and FFs.

† Values estimated by multiplying time by frequency.

^{4} ^{6} indicates LUT size.

$\clubsuit$ BRAMs of Virtex 5, 6 and Zynq are 1k×36, so they account as 2 independent 1k×18.

Table 5. Comparison of our results to the literature on hardware implementations for ECC for 256 bits field. The speed results are for one scalar multiplication.

4. A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache. Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, Nov 2012. 6
5. L. Batina, Ł. Chmielewski, L. Papachristodoulou, P. Schwabe, and M. Tunstall. Online Template Attacks. In *Progress in Cryptology – INDOCRYPT 2014: 15th International Conference on Cryptology in India, New Delhi, India, December 14-17, 2014, Proceedings*, pages 21–36. Springer International Publishing, 2014. 3
6. D. J. Bernstein. Curve25519: new Diffie-Hellman speed records. In *Public key cryptography—PKC 2006, 9th international conference on theory and practice in public-key cryptography*, New York, NY, USA, 2006. Springer. 1
7. D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters. Twisted Edwards Curves. In *Progress in Cryptology - AFRICACRYPT 2008, First International Conference on Cryptology in Africa, Casablanca, Morocco, June 11-14, 2008. Proceedings*, pages 389–405, 2008. 1
8. D. J. Bernstein, C. Chuengsatiansup, D. Kohel, and T. Lange. Twisted Hessian Curves. In *Progress in Cryptology - LATINCRYPT 2015 - 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings*, pages 269–294, 2015. 1
9. D. J. Bernstein and T. Lange. Faster Addition and Doubling on Elliptic Curves. In *Advances in Cryptology – ASIACRYPT 2007: 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2-6, 2007. Proceedings*, pages 29–50, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. 1
10. D. J. Bernstein and T. Lange. Explicit-Formulas Database. http://hyperelliptic.org/EFD/index.html, Date accessed: February 21, 2015. 3
11. D. J. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. In A. Hevia and G. Neven, editors, *Progress in Cryptology – LATINCRYPT 2012*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176. Springer-Verlag Berlin Heidelberg, 2012. Document ID: 5f6fc69cc5a319aecba43760c56fab04, http://cryptojedi.org/papers/#coolnacl. 1
12. W. Bosma, J. J. Cannon, and C. Playoust. The Magma algebra system I: the user language. *J. Symb. Comput.*, 24(3/4):235–265, 1997. 2
13. Certicom Research. SEC 2: Recommended Elliptic Curve Domain Parameters, Version 2.0. Technical report, Certicom Research, 2010. 1
14. C. Clavier and M. Joye. Universal Exponentiation Algorithm - A First Step towards Provable SPA-Resistance. In *Cryptographic Hardware and Embedded Systems - CHES 2001, volume 2162 of Lecture Notes in Computer Science*, pages 300–308. Springer-Verlag, 2001. 7
15. J.-S. Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, pages 292–302, 1999. 3, 4, 6, 7
16. ECC Brainpool. ECC Brainpool standard curves and curve generation. Technical report, Brainpool, 2005. 1
17. J. Fan, K. Sakiyama, and I. Verbauwhede. Elliptic curve cryptography on embedded multicore systems. *Design Automation for Embedded Systems*, 12(3):231–242, 2008. 2, 14
18. S. D. Galbraith. *Mathematics of Public Key Cryptography.* Cambridge University Press, 2012. 3
19. T. Güneysu and C. Paar. Ultra High Performance ECC over NIST Primes on Commercial FPGAs. In *Cryptographic Hardware and Embedded Systems – CHES 2008*, volume 5154 of *Lecture Notes in Computer Science*, pages 62–78. Springer Berlin Heidelberg, 2008. 2, 10, 11, 14
20. N. Guillermin. A high speed coprocessor for elliptic curve scalar multiplications over $\mathbb{F}_p$. In *Cryptographic Hardware and Embedded Systems, CHES 2010: 12th International Workshop, Santa Barbara, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 48–64. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. 2, 10, 11, 14
21. M. Hamburg. Ed448-Goldilocks, a new elliptic curve, 2015. http://eprint.iacr.org/2015/625.pdf. 1
22. M. Joye and S. Yen. The Montgomery Powering Ladder. In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, pages 291–302, 2002. 4
23. Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *Micro, IEEE*, 16(3):26–33, Jun 1996. 6
24. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987. 1, 3
25. P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Advances in Cryptology – CRYPTO' 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer Berlin Heidelberg, 1999. 1, 3
26. 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*, 23(11):2753–2756, Nov 2015. 10, 11, 14

27. Y. Ma, Z. Liu, W. Pan, and J. Jing. A High-Speed Elliptic Curve Cryptographic Processor for Generic Curves over GF(p). In *Selected Areas in Cryptography – SAC 2013: 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, pages 421–437, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. 10, 11, 14
28. P. M. C. Massolino, L. Batina, R. Chaves, and N. Mentens. Low Power Montgomery Modular Multiplication on Reconfigurable Systems. Cryptology ePrint Archive, Report 2016/280, 2016. `http://eprint.iacr.org/2016/280`. 2, 5, 6, 7, 8, 9
29. C. McIvor, M. McLoone, and J. V. McCanny. Hardware Elliptic Curve Cryptographic Processor Over GF(p). *IEEE Transactions on Circuits and Systems I: Regular Papers*, 53(9):1946–1957, Sept 2006. 10, 11, 14
30. V. Miller. Use of Elliptic Curves in Cryptography. In *Advances in Cryptology - CRYPTO 85 Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer Berlin / Heidelberg, Berlin, Germany, 1986. 1, 3
31. National Institute for Standards and Technology. Federal information processing standards publication 186-4. digital signature standard. Technical report, NIST, 2013. 1, 7, 11, 14
32. C. Pöpper, O. Mischke, and T. Güneysu. MicroACP - A Fast and Secure Reconfigurable Asymmetric Crypto-Processor. In *Reconfigurable Computing: Architectures, Tools, and Applications*, volume 8405 of *Lecture Notes in Computer Science*, pages 240–247. Springer International Publishing, 2014. 2, 11, 14
33. J. Renes, C. Costello, and L. Batina. Complete addition formulas for prime order elliptic curves. In *Advances in Cryptology – EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, pages 403–428, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. 2, 3, 4, 5, 10
34. D. B. Roy, P. Das, and D. Mukhopadhyay. ECC on Your Fingertips: A Single Instruction Approach for Lightweight ECC Design in GF (p). In *Selected Areas in Cryptography - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers*. Springer International Publishing, 2015. 2, 10, 11, 14
35. K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede. Superscalar Coprocessor for High-speed Curve-based Cryptography. In *Proceedings of the 8th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'06, pages 415–429, Berlin, Heidelberg, 2006. Springer-Verlag. 2
36. P. Sasdrich and T. Güneysu. Efficient Elliptic-Curve Cryptography Using Curve25519 on Reconfigurable Devices. In *Reconfigurable Computing: Architectures, Tools, and Applications*, volume 8405 of *Lecture Notes in Computer Science*, pages 25–36. Springer International Publishing, 2014. 10, 11, 14
37. J. H. Silverman. *The Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics. Springer-Verlag New York, 2009. 3
38. M. Varchola, T. Güneysu, and O. Mischke. MicroECC: A Lightweight Reconfigurable Elliptic Curve Crypto-processor. In *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pages 204–210, Nov 2011. 2, 10, 11, 14
39. J. Vliegen, N. Mentens, J. Genoe, A. Braeken, S. Kubera, A. Touhafi, and I. Verbauwhede. A compact FPGA-based architecture for elliptic curve cryptography over prime fields. In *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pages 313–316, July 2010. 3, 10, 11, 14
40. G. X. Yao, J. Fan, R. C. C. Cheung, and I. Verbauwhede. Faster Pairing Coprocessor Architecture. In *Pairing-Based Cryptography – Pairing 2012: 5th International Conference, Cologne, Germany, May 16-18, 2012, Revised Selected Papers*, pages 160–176, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 2
41. S. Yen and M. Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers*, 49(9):967–970, Sep 2000. 1

A More complete results comparison

Work	Field	FPGA	Slice/	LUT	FF	Emb.	BRAM	BRAM	Freq.	Scalar Mult.	
			ALM			Mult.	64×18	1k×18	(MHz)	Cycles	(ms)
For all prime fields and prime order short Weierstrass curves											
Our	192	IGLOO 2^4	–	2828	1048	6	6	1	165	728508	4.42
Our	224	IGLOO 2^4	–	2828	1048	6	6	1	165	1036294	6.28
Our	256	IGLOO 2^4	–	2828	1048	6	6	1	165	1421392	8.61
Our	320	IGLOO 2^4	–	2828	1048	6	6	1	165	2498655	15.14
Our	384	IGLOO 2^4	–	2828	1048	6	6	1	165	3744883	22.70
Our	512	IGLOO 2^4	–	2828	1048	6	6	1	165	8188059	49.62
Our	521	IGLOO 2^4	–	2828	1048	6	6	1	165	8331987	50.50
For NIST curves [31] only											
[38]	224	SmartFusion^4	–	3690	3690	0	0	12	109	1722088	15.8
[38]	256	SmartFusion^4	–	3690	3690	0	0	12	109	2103941	19.3
[38]	224	Virtex II Pro^4	773	*1546	*1546	1	0	3	210	1722088	8.2
[38]	256	Virtex II Pro^4	773	*1546	*1546	1	0	3	210	2103941	10.02
[38]	224	Virtex II Pro^4	1158	*2316	*2316	4	0	3	210	765072	3.64
[38]	256	Virtex II Pro^4	1158	*2316	*2316	4	0	3	210	949951	4.52
[32]	256	Virtex $5^{6\clubsuit}$	1914	*7656	*7656	4	0	12	210	830000	3.95
[17]	192	Virtex II Pro^4	3173	*6346	*6346	16	0	6	93	†920700	9.90
[34]	256	Spartan 6^6	72	193	35	8	0	24	156.25	†1906250	12.2
[26]	192	Virtex 4^4	7020	12435	3545	8	0	4	182	†429702	2.361
[26]	224	Virtex 4^4	7020	12435	3545	8	0	4	182	†666666	3.663
[26]	256	Virtex 4^4	7020	12435	3545	8	0	4	182	†993174	5.457
[26]	384	Virtex 4^4	7020	12435	3545	8	0	4	182	†2968420	16.31
[26]	521	Virtex 4^4	7020	12435	3545	8	0	4	182	†7048860	38.73
[1]	192	Virtex $6^{6\clubsuit}$	11.2k	32.9k	*89.6k	289	0	256	100	29948	0.30
[1]	224	Virtex $6^{6\clubsuit}$	11.2k	32.9k	*89.6k	289	0	256	100	34999	0.35
[1]	256	Virtex $6^{6\clubsuit}$	11.2k	32.9k	*89.6k	289	0	256	100	39922	0.40
[1]	384	Virtex $6^{6\clubsuit}$	11.2k	32.9k	*89.6k	289	0	256	100	11722	1.18
[1]	521	Virtex $6^{6\clubsuit}$	11.2k	32.9k	*89.6k	289	0	256	100	159959	1.60
[19]	224	Virtex 4^4	1580	1825	1892	26	0	11	487	219878	0.451
[19]	256	Virtex 4^4	1715	2589	2028	32	0	11	490	303450	0.619
For only Edwards or Twisted Edwards curves											
[3]	192	Spartan 3E ^{4}	4654	*9308	*9308	0	0	0	10	125430†	12.543
[36]	256	Zynq$^{6\clubsuit}$	1029	2783	3592	20	0	4	200	64770	0.324
For only specific field size, but works with any prime											
[39]	256	Virtex II Pro^4	1832	*3664	*3664	2	0	9	108.2	3227993	29.83
[39]	256	Virtex II Pro^4	2085	*4170	*4170	7	0	9	68.17	1074625	15.76
[20]	192	Stratix II^4	6203	*12406	*12406	92	0	0	160.5	†70620	0.44
[20]	256	Stratix II^4	9177	*18354	*18354	96	0	0	157.2	†106896	0.68
[20]	384	Stratix II^4	12958	*25916	*25916	177	0	0	150.9	†203715	1.35
[20]	512	Stratix II^4	17017	*34034	*34034	244	0	0	144.97	†323283	2.23
[29]	256	Virtex II Pro^4	15755	*31510	*31510	256	0	0	39.46	†151360	3.86
[27]	256	Virtex 4^4	4655	5740	4876	37	0	11	250	109297	0.44
[2]	192	Virtex $5^{6\clubsuit}$	1769	6096	4370	0	0	0	96.57	198854	2.05
[2]	256	Virtex $5^{6\clubsuit}$	2284	7822	5780	0	0	0	81.71	†331200	4.04

Table 6. Complete comparison and results from Table 5. This table does not have the results for other FPGAs, since they are are the same as in Table 5, the difference is the cycle count for each field.

B Algorithms

Algorithm 2 Parallelized complete addition formulas for a prime order elliptic curve in Weierstrass form, using *two* processors

Require: $P = (X_1 : Y_1 : Z_1)$, $Q = (X_2 : Y_2 : Z_2)$, $E\colon Y^2Z = X^3 + aXZ^2 + bZ^3$ and $b_3 = 3 \cdot b$.

Ensure: $(X_3 : Y_3 : Z_3) = P + Q$.

```
$t_0 \leftarrow X_1 + Y_1;$                         $t_1 \leftarrow X_2 + Y_2;$
$t_2 \leftarrow Y_1 + Z_1;$                         $t_3 \leftarrow Y_2 + Z_2;$
$t_0 \leftarrow t_0 \cdot t_1; (\ell_3)$            $t_1 \leftarrow t_2 \cdot t_3; (\ell_5)$
$t_4 \leftarrow X_1 \cdot X_2; (\ell_0)$            $t_6 \leftarrow Z_1 \cdot Z_2; (\ell_2)$
$t_2 \leftarrow X_1 + Z_1;$                         $t_3 \leftarrow X_2 + Z_2;$
$t_0 \leftarrow t_0 - t_4;$                         $t_1 \leftarrow t_1 - t_6;$
$t_5 \leftarrow Y_1 \cdot Y_2; (\ell_1)$            $t_2 \leftarrow t_2 \cdot t_3; (\ell_4)$
$t_7 \leftarrow a \cdot t_6; (\ell_7)$              $t_8 \leftarrow b_3 \cdot t_6; (\ell_8)$
$t_9 \leftarrow t_4 - t_7;$                         $t_{10} \leftarrow t_4 + t_4;$
$t_{11} \leftarrow t_4 + t_7;$                      $t_2 \leftarrow t_2 - t_4;$
$t_0 \leftarrow t_0 - t_5;$                         $t_1 \leftarrow t_1 - t_5;$
$t_2 \leftarrow t_2 - t_6;$                         $t_{10} \leftarrow t_{10} + t_{11};$
$t_9 \leftarrow a \cdot t_9; (\ell_{10})$           $t_{11} \leftarrow b_3 \cdot t_2; (\ell_9)$
$t_2 \leftarrow a \cdot t_2; (\ell_8)$
$t_9 \leftarrow t_9 + t_{11};$                      $t_8 \leftarrow t_2 + t_8;$
$t_6 \leftarrow t_5 - t_8;$                         $t_5 \leftarrow t_5 + t_8;$
$t_3 \leftarrow t_1 \cdot t_9; (\ell_{12})$         $t_9 \leftarrow t_9 \cdot t_{10}; (\ell_{14})$
$t_{10} \leftarrow t_0 \cdot t_{10}; (\ell_{16})$   $t_0 \leftarrow t_0 \cdot t_6; (\ell_{11})$
$t_6 \leftarrow t_5 \cdot t_6; (\ell_{13})$         $t_1 \leftarrow t_1 \cdot t_5; (\ell_{15})$
$X_3 \leftarrow t_0 - t_3;$                         $Y_9 \leftarrow t_6 + t_9;$
$Z_3 \leftarrow t_1 + t_{10};$
```

Algorithm 3 Parallelized complete addition formulas for a prime order elliptic curve in Weierstrass form, using *three* processors

Require: $P = (X_1 : Y_1 : Z_1)$, $Q = (X_2 : Y_2 : Z_2)$, $E\colon Y^2Z = X^3 + aXZ^2 + bZ^3$ and $b_3 = 3 \cdot b$.

Ensure: $(X_3 : Y_3 : Z_3) = P + Q$.

```
1.  $t_0 \leftarrow X_1 \cdot X_2; (\ell_0)$    $t_1 \leftarrow Y_1 \cdot Y_2; (\ell_1)$    $t_2 \leftarrow Z_1 \cdot Z_2; (\ell_2)$
2.  $t_3 \leftarrow X_1 + Y_1;$    $t_4 \leftarrow X_2 + Y_2;$    $t_5 \leftarrow Y_1 + Z_1;$
3.  $t_6 \leftarrow Y_2 + Z_2;$    $t_7 \leftarrow X_1 + Z_1;$    $t_8 \leftarrow X_2 + Z_2;$
4.  $t_9 \leftarrow t_3 \cdot t_4; (\ell_3)$    $t_{10} \leftarrow t_5 \cdot t_6; (\ell_5)$    $t_{11} \leftarrow t_7 \cdot t_8; (\ell_4)$
5.  $t_3 \leftarrow t_0 + t_1;$    $t_4 \leftarrow t_1 + t_2;$    $t_5 \leftarrow t_0 + t_2;$
6.  $t_6 \leftarrow b_3 \cdot t_2; (\ell_6)$    $t_8 \leftarrow a \cdot t_2; (\ell_7)$
7.  $t_2 \leftarrow t_9 - t_3;$    $t_9 \leftarrow t_0 + t_0;$    $t_3 \leftarrow t_{10} - t_4;$
8.  $t_{10} \leftarrow t_9 + t_0;$    $t_4 \leftarrow t_{11} - t_5;$    $t_7 \leftarrow t_0 - t_8;$
9.  $t_0 \leftarrow a \cdot t_4; (\ell_8)$    $t_5 \leftarrow b_3 \cdot t_4; (\ell_9)$    $t_9 \leftarrow a \cdot t_7; (\ell_{10})$
10. $t_4 \leftarrow t_0 + t_6;$    $t_7 \leftarrow t_5 + t_9;$    $t_0 \leftarrow t_8 + t_{10};$
11. $t_5 \leftarrow t_1 - t_4;$    $t_6 \leftarrow t_1 + t_4;$
12. $t_1 \leftarrow t_5 \cdot t_6; (\ell_{13})$    $t_4 \leftarrow t_0 \cdot t_7; (\ell_{14})$    $t_8 \leftarrow t_3 \cdot t_7; (\ell_{12})$
13. $t_9 \leftarrow t_2 \cdot t_5; (\ell_{11})$    $t_{10} \leftarrow t_3 \cdot t_6; (\ell_{15})$    $t_{11} \leftarrow t_0 \cdot t_2; (\ell_{16})$
14. $X_3 \leftarrow t_9 - t_8;$    $Y_3 \leftarrow t_1 + t_4;$    $Z_3 \leftarrow t_{10} + t_{11};$
```

Algorithm 4 Parallelized complete addition formulas for a prime order elliptic curve in Weierstrass form, using *four* processors

Require: $P = (X_1 : Y_1 : Z_1)$, $Q = (X_2 : Y_2 : Z_2)$, $E\colon Y^2Z = X^3 + aXZ^2 + bZ^3$ and $b_3 = 3 \cdot b$.

Ensure: $(X_3 : Y_3 : Z_3) = P + Q$.

```
1.  $t_0 \leftarrow X_1 + Y_1;$    $t_1 \leftarrow X_2 + Y_2;$    $t_2 \leftarrow Y_1 + Z_1;$    $t_3 \leftarrow Y_2 + Z_2;$
2.  $t_0 \leftarrow t_0 \cdot t_1; (\ell_3)$    $t_1 \leftarrow t_2 \cdot t_3; (\ell_5)$    $t_4 \leftarrow X_1 \cdot X_2; (\ell_0)$    $t_6 \leftarrow Z_1 \cdot Z_2; (\ell_2)$
3.  $t_2 \leftarrow X_1 + Z_1;$    $t_3 \leftarrow X_2 + Z_2;$    $t_0 \leftarrow t_0 - t_4;$    $t_1 \leftarrow t_1 - t_6;$
4.  $t_5 \leftarrow Y_1 \cdot Y_2; (\ell_1)$    $t_2 \leftarrow t_2 \cdot t_3; (\ell_4)$    $t_7 \leftarrow a \cdot t_6; (\ell_7)$    $t_8 \leftarrow b_3 \cdot t_6; (\ell_6)$
5.  $t_9 \leftarrow t_4 - t_7;$    $t_{10} \leftarrow t_4 + t_4;$    $t_{11} \leftarrow t_4 + t_7;$    $t_2 \leftarrow t_2 - t_4;$
6.  $t_0 \leftarrow t_0 - t_5;$    $t_1 \leftarrow t_1 - t_5;$    $t_2 \leftarrow t_2 - t_6;$    $t_{10} \leftarrow t_{10} + t_{11};$
7.  $t_9 \leftarrow a \cdot t_9; (\ell_{10})$    $t_{11} \leftarrow b_3 \cdot t_2; (\ell_9)$    $t_2 \leftarrow a \cdot t_2; (\ell_8)$
8.  $t_9 \leftarrow t_9 + t_{11};$
9.  $t_3 \leftarrow t_1 \cdot t_9; (\ell_{12})$    $t_9 \leftarrow t_9 \cdot t_{10}; (\ell_{14})$    $t_{10} \leftarrow t_0 \cdot t_{10}; (\ell_{16})$    $t_8 \leftarrow t_2 + t_8;$
10. $t_6 \leftarrow t_5 - t_8;$    $t_5 \leftarrow t_5 + t_8;$
11. $t_0 \leftarrow t_0 \cdot t_6; (\ell_{11})$    $t_6 \leftarrow t_5 \cdot t_6; (\ell_{13})$    $t_1 \leftarrow t_1 \cdot t_5; (\ell_{15})$
12. $X_3 \leftarrow t_0 - t_3;$    $Y_3 \leftarrow t_6 + t_9;$    $Z_3 \leftarrow t_1 + t_{10};$
```

Algorithm 5 Parallelized complete addition formulas for a prime order elliptic curve in Weierstrass form, using *five* processors

```
Require: $P = (X_1 : Y_1 : Z_1)$, $Q = (X_2 : Y_2 : Z_2)$, $E\colon Y^2Z = X^3 + aXZ^2 + bZ^3$ and $b_3 = 3 \cdot b$.
Ensure: $(X_3 : Y_3 : Z_3) = P + Q$.
 1. $t_5 \leftarrow X_1 + Y_1;$   $t_6 \leftarrow X_2 + Y_2;$   $t_7 \leftarrow X_1 + Z_1;$
    $t_8 \leftarrow X_2 + Z_2;$   $t_9 \leftarrow Y_1 + Z_1;$
 2. $t_0 \leftarrow X_1 \cdot X_2; (\ell_0)$   $t_1 \leftarrow Y_1 \cdot Y_2; (\ell_1)$   $t_2 \leftarrow Z_1 \cdot Z_2; (\ell_2)$
    $t_3 \leftarrow t_5 \cdot t_6; (\ell_3)$   $t_4 \leftarrow t_7 \cdot t_8; (\ell_4)$
 3. $t_{10} \leftarrow Y_2 + Z_2;$   $t_3 \leftarrow t_3 - t_0;$   $t_4 \leftarrow t_4 - t_0;$
    $t_{11} \leftarrow t_0 + t_0;$
 4. $t_3 \leftarrow t_3 - t_1;$   $t_4 \leftarrow t_4 - t_2;$   $t_{11} \leftarrow t_{11} + t_0;$
 5. $t_5 \leftarrow t_9 \cdot t_{10}; (\ell_5)$   $t_6 \leftarrow b_3 \cdot t_2; (\ell_6)$   $t_7 \leftarrow a \cdot t_2; (\ell_7)$
    $t_8 \leftarrow a \cdot t_4; (\ell_8)$   $t_9 \leftarrow b_3 \cdot t_4; (\ell_9)$
 6. $t_5 \leftarrow t_5 - t_1;$   $t_{11} \leftarrow t_{11} + t_7;$   $t_4 \leftarrow t_0 - t_7;$
    $t_{10} \leftarrow t_6 + t_8;$
 7. $t_0 \leftarrow a \cdot t_4; (\ell_{10})$   $t_6 \leftarrow t_3 \cdot t_{11}; (\ell_{16})$
 8. $t_0 \leftarrow t_0 + t_9;$   $t_7 \leftarrow t_1 - t_{10};$   $t_{10} \leftarrow t_1 + t_{10};$
    $t_5 \leftarrow t_5 - t_2;$
 9. $t_1 \leftarrow t_3 \cdot t_7; (\ell_{11})$   $t_2 \leftarrow t_5 \cdot t_0; (\ell_{12})$   $t_4 \leftarrow t_{10} \cdot t_7; (\ell_{13})$
    $t_8 \leftarrow t_{11} \cdot t_0; (\ell_{14})$   $t_9 \leftarrow t_5 \cdot t_{10}; (\ell_{15})$
10. $X_3 \leftarrow t_1 - t_2;$   $Y_3 \leftarrow t_4 + t_8;$   $Z_3 \leftarrow t_9 + t_6;$
```

Algorithm 6 Parallelized complete addition formulas for a prime order elliptic curve in Weierstrass form, using *six* processors

```
Require: $P = (X_1 : Y_1 : Z_1)$, $Q = (X_2 : Y_2 : Z_2)$, $E\colon Y^2Z = X^3 + aXZ^2 + bZ^3$, $b_3 = 3 \cdot b$ and $a_2 = a^2$.
Ensure: $(X_3 : Y_3 : Z_3) = P + Q$.
1. $t_0 \leftarrow X_1 + Y_1;$                $t_1 \leftarrow X_2 + Y_2;$              $t_2 \leftarrow Y_1 + Z_1;$
   $t_3 \leftarrow Y_2 + Z_2;$                $t_4 \leftarrow X_1 + Z_1;$              $t_5 \leftarrow X_2 + Z_2;$
2. $t_0 \leftarrow t_0 \cdot t_1; (n_3)$      $t_1 \leftarrow t_2 \cdot t_3; (n_5)$    $t_2 \leftarrow t_4 \cdot t_5; (n_4)$
   $t_3 \leftarrow X_1 \cdot X_2; (n_0)$      $t_4 \leftarrow Y_1 \cdot Y_2; (n_1)$    $t_5 \leftarrow Z_1 \cdot Z_2; (n_2)$
3. $t_0 \leftarrow t_0 - t_3;$                $t_1 \leftarrow t_1 - t_4;$              $t_2 \leftarrow t_2 - t_5;$
4. $t_0 \leftarrow t_0 - t_4;$                $t_1 \leftarrow t_1 - t_5;$              $t_2 \leftarrow t_2 - t_3;$
5. $t_6 \leftarrow b_3 \cdot t_5; (n_6)$      $t_7 \leftarrow a \cdot t_5; (n_7)$      $t_8 \leftarrow a \cdot t_2; (n_8)$
   $t_9 \leftarrow b_3 \cdot t_2; (n_9)$      $t_{10} \leftarrow a \cdot t_3; (n_{10})$  $t_{11} \leftarrow a_2 \cdot t_5; (n_{11})$
6. $t_6 \leftarrow t_6 + t_8;$                $t_7 \leftarrow t_3 + t_7$               $t_8 \leftarrow t_3 + t_3;$
   $t_9 \leftarrow t_9 + t_{10};$
7. $t_9 \leftarrow t_9 - t_{11};$             $t_8 \leftarrow t_8 + t_7$               $t_7 \leftarrow t_4 - t_6;$
   $t_6 \leftarrow t_4 + t_6;$
8. $t_3 \leftarrow t_0 \cdot t_7; (n_{12})$   $t_4 \leftarrow t_0 \cdot t_8; (n_{17})$  $t_5 \leftarrow t_1 \cdot t_9; (n_{13})$
   $t_8 \leftarrow t_8 \cdot t_9; (n_{15})$   $t_7 \leftarrow t_6 \cdot t_7; (n_{14})$  $t_6 \leftarrow t_1 \cdot t_6; (n_{16})$
9. $X_3 \leftarrow t_3 - t_5;$                $Y_3 \leftarrow t_7 + t_8;$              $Z_3 \leftarrow t_6 + t_4;$
```

C Verification code

```
ADD_two:=function(X1,Y1,Z1,X2,Y2,Z2,E,a,b3)
    t0 := X1+Y1;   t1 := X2+Y2;
    t2 := Y1+Z1;   t3 := Y2+Z2;
    t0 := t0*t1;   t1 := t2*t3;
    t4 := X1*X2;   t6 := Z1*Z2;
    t2 := X1+Z1;   t3 := X2+Z2;
    t0 := t0-t4;   t1 := t1-t6;
    t5 := Y1*Y2;   t2 := t2*t3;
    t7 := a*t6;    t8 := b3*t6;
    t9 := t4-t7;   t10 := t4+t4;
    t11 := t4+t7;  t2 := t2-t4;
    t0 := t0-t5;   t1 := t1-t5;
    t2 := t2-t6;   t10 := t10+t11;
    t9 := a*t9;    t11 := b3*t2;
    t2 := a*t2;
    t9 := t9+t11;  t8 := t2+t8;
    t6 := t5-t8;   t5 := t5+t8;
    t3 := t1*t9;   t9 := t9*t10;
    t10 := t0*t10; t0 := t0*t6;
    t6 := t5*t6;   t1 := t1*t5;
    X3 := t0-t3;   Y3 := t6+t9;
    Z3 := t1+t10;
    return E![X3,Y3,Z3];
end function;

ADD_three:=function(X1,Y1,Z1,X2,Y2,Z2,E,a,b3);
    t0 := X1*X2;   t1 := Y1*Y2;   t2 := Z1*Z2;
    t3 := X1+Y1;   t4 := X2+Y2;   t5 := Y1+Z1;
    t6 := Y2+Z2;   t7 := X1+Z1;   t8 := X2+Z2;
    t9 := t3*t4;   t10 := t5*t6;  t11 := t7*t8;
    t3 := t0+t1;   t4 := t1+t2;   t5 := t0+t2;
    t6 := b3*t2;   t8 := a*t2;
    t2 := t9-t3;   t9 := t0+t0;   t3 := t10-t4;
    t10 := t9+t0;  t4 := t11-t5;  t7 := t0-t8;
    t0 := a*t4;    t5 := b3*t4;   t9 := a*t7;
    t4 := t0+t6;   t7 := t5+t9;   t0 := t8+t10;
    t5 := t1-t4;   t6 := t1+t4;
    t1 := t5*t6;   t4 := t0*t7;   t8 := t3*t7;
    t9 := t2*t5;   t10 := t3*t6;  t11 := t0*t2;
    X3 := t9-t8;   Y3 := t1+t4;   Z3 := t10+t11;
    return E![X3,Y3,Z3];
end function;

ADD_four:=function(X1,Y1,Z1,X2,Y2,Z2,E,a,b3);
   t0 := X1+Y1;  t1 := X2+Y2;  t2 := Y1+Z1;   t3 := Y2+Z2;
   t0 := t0*t1;  t1 := t2*t3;  t4 := X1*X2;   t6 := Z1*Z2;
   t2 := X1+Z1;  t3 := X2+Z2;  t0 := t0-t4;   t1 := t1-t6;
   t5 := Y1*Y2;  t2 := t2*t3;  t7 := a*t6;;   t8 := b3*t6;
   t9 := t4-t7;  t10 := t4+t4; t11 := t4+t7;  t2 := t2-t4;
   t0 := t0-t5;  t1 := t1-t5;  t2 := t2-t6;   t10 := t10+t11;
   t9 := a*t9;   t11 := b3*t2; t2 := a*t2;
   t9 := t9+t11;
   t3 := t1*t9;  t9 := t9*t10; t10 := t0*t10; t8 := t2+t8;
   t6 := t5-t8;  t5 := t5+t8;
   t0 := t0*t6;  t6 := t5*t6;  t1 := t1*t5;
   X3 := t0-t3;  Y3 := t6+t9;  Z3 := t1+t10;
   return E![X3,Y3,Z3];
end function;

ADD_five:=function(X1,Y1,Z1,X2,Y2,Z2,E,a,b3);
    t5 := X1+Y1;   t6 := X2+Y2;   t7 := X1+Z1;
    t8 := X2+Z2;   t9 := Y1+Z1;                  // 1
    t0 := X1*X2;   t1 := Y1*Y2;   t2 := Z1*Z2;
    t3 := t5*t6;   t4 := t7*t8;                  // 2
    t10 := Y2+Z2;  t3 := t3-t0;   t4 := t4-t0;
    t11 := t0+t0;                                // 3
    t3 := t3-t1;   t4 := t4-t2;   t11 := t11+t0; // 4
    t5 := t9*t10;  t6 := b3*t2;   t7 := a*t2;
    t8 := a*t4;    t9 := b3*t4;                  // 5
    t5 := t5-t1;   t11 := t11+t7; t4 := t0-t7;
    t10 := t6+t8;                                // 6
    t0 := a*t4;    t6 := t3*t11;                 // 7
    t0 := t0+t9;   t7 := t1-t10;  t10 := t1+t10;
    t5 := t5-t2;                                 // 8
    t1 := t3*t7;   t2 := t5*t0;   t4 := t10*t7;
    t8 := t11*t0;  t9 := t5*t10;                 // 9
    X3 := t1-t2;   Y3 := t4+t8;   Z3 := t9+t6;   // 10
    return E![X3,Y3,Z3];
end function;

ADD_six:=function(X1,Y1,Z1,X2,Y2,Z2,E,a,b3)
    t0 := X1+Y1;   t1 := X2+Y2;   t2 := Y1+Z1;
    t3 := Y2+Z2;   t4 := X1+Z1;   t5 := X2+Z2;   // 1
    t0 := t0*t1;   t1 := t2*t3;   t2 := t4*t5;
    t3 := X1*X2;   t4 := Y1*Y2;   t5 := Z1*Z2;   // 2
    t0 := t0-t3;   t1 := t1-t4;   t2 := t2-t5;   // 3
    t0 := t0-t4;   t1 := t1-t5;   t2 := t2-t3;   // 4
    t6 := b3*t5;   t7 := a*t5;    t8 := a*t2;
    t9 := b3*t2;   t10 := a*t3;   t11 := a^2*t5; // 5
    t6 := t6+t8;   t7 := t3+t7;   t8 := t3+t3;
    t9 := t9+t10;                                // 6
    t9 := t9-t11;  t8 := t8+t7;   t7 := t4-t6;
    t6 := t4+t6;                                 // 7
    t3 := t0*t7;   t4 := t0*t8;   t5 := t1*t9;
    t8 := t8*t9;   t7 := t6*t7;   t6 := t1*t6;   // 8
    X3 := t3-t5;   Y3 := t7+t8;   Z3 := t6+t4;   // 9
    return E![X3,Y3,Z3];
end function;

while(true) do
    repeat q:=RandomPrime(8); until q gt 3;
    Fq:=GF(q);
    repeat repeat a:=Random(Fq); b:=Random(Fq); until not (4*a^3+27*b^2 eq 0);
        E:=EllipticCurve([Fq|a,b]);
        b3 := 3*b;
    until IsOdd(#E);

    for P in Set(E) do
        for Q in Set(E) do
           repeat Z1 := Random(Fq); until Z1 ne 0;
           repeat Z2 := Random(Fq); until Z2 ne 0;
           X1 := P[1]*Z1;   Y1 := P[2]*Z1;  Z1 := P[3]*Z1;
           X2 := Q[1]*Z2;   Y2 := Q[2]*Z2;  Z2 := Q[3]*Z2;
```

```
            assert P+Q eq ADD_two(X1,Y1,Z1,X2,Y2,Z2,E,a,b3);
            assert P+Q eq ADD_three(X1,Y1,Z1,X2,Y2,Z2,E,a,b3);
            assert P+Q eq ADD_four(X1,Y1,Z1,X2,Y2,Z2,E,a,b3);
            assert P+Q eq ADD_five(X1,Y1,Z1,X2,Y2,Z2,E,a,b3);
            assert P+Q eq ADD_six(X1,Y1,Z1,X2,Y2,Z2,E,a,b3);
        end for;
    end for;
    print "Correct:", E;
end while;
```