Implementing Complete Formulas on Weierstrass Curves in Hardware

Pedro Maat C. Massolino(✉), Joost Renes, and Lejla Batina

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 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 [22,28]. This resulted in more optimized implementations, since doublings can be faster than general additions, but naïve implementations suffered from side-channel attacks [23]. 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 avoids these problems is the family of curves proposed by Bernstein and Lange, the so-called Edwards curves [8]. 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 thwarts side-channel analysis and more refined attacks, e. g. safe-error attacks [38]. After the introduction of Edwards curves, more curves models have been shown to possess complete addition laws [6,7].

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

Moreover, (twisted) Edwards curves are being deployed in software, for example in the library NaCl [10]. In particular, software implementations typically rely on specific curves, e. g. on the Montgomery curves Curve25519 [5] by Bernstein or Curve448 [19] 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 most current standards [12,15,29] regarding large prime fields contain prime order curves in short Weierstrass form. This prohibits using (twisted) Edwards, (twisted) Hessian and Montgomery curves. The desire for complete addition formulas for prime order curves in short Weierstrass form was recognized and Renes, Costello and Batina [31] proved this to be realistic. They present complete addition formulas with an efficiency loss of 34 %–44 % in software when compared to formulas based on Jacobian coordinates, depending on the size of the field.

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 [26], 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 to boost the performance.

Related Work. Mainly there are numerous works on curve-based hardware implementations. These are on various FPGA platforms, making a meaningful comparison very difficult. Güneysu and Paar [17] proposed a new speed-optimized architecture that makes intensive use of the DSP blocks in an FPGA platform. Guillermin [18] 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. [37] followed the idea of using RNS to design a high-speed ECC co-processor for pairings. Sakiyama et al. [33] proposed a superscalar coprocessor that could deal with three different curve-based cryptosystems, all in characteristic 2 fields. Varchola et al. [35] 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 was the case for Güneysu and Paar [17], 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. [30] follow the same approach as Varchola et al. [35], with some side-channel related improvements. The paper focuses on an analysis of each countermeasure and its effective cost. Roy et al. [32] 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. [30] or Varchola et al. [35]. Fan et al. [16] 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. [36] attempted to reduce the resources with a small core aimed at 256-bit primes.

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

2 Preliminaries for Elliptic Curve Cryptography

Let $\mathbb{F}_q$ be a finite field of characteristic p, i. e. $q = p^n$ for some n, and assume that p is not two or three. For well-chosen $a, b \in \mathbb{F}_q$, an *elliptic curve* E over $\mathbb{F}_q$ is defined as the set of solutions (x, y) to the curve equation $E : y^2 = x^3 + ax + b$ with an additional point $\mathcal{O}$, called the *point at infinity.* The $\mathbb{F}_q$-rational points $E(\mathbb{F}_q)$ are all $(x, y) \in E$ such that $(x, y) \in \mathbb{F}_q^2$, together with $\mathcal{O}$. They form a group, with $\mathcal{O}$ as its identity element. From now on when we write E, we mean $E(\mathbb{F}_q)$. The *order* of E is the order of this group. To compute the group law on E one can use the chord and tangent process. To implement this, however, it is necessary to use at least one inversion. Since inversions are very costly, we choose a different point representation to avoid them.

Define an equivalence relation on $\mathbb{F}_q^3$ by letting $(x_0, x_1, x_2) \sim (y_0, y_1, y_2)$ if and only if there exists $\lambda \in \mathbb{F}_q^*$ such that $(x_0, x_1, x_2) = (\lambda y_0, \lambda y_1, \lambda y_2)$. Then the *projective plane* over $\mathbb{F}_q$, denoted $\mathbb{P}^2(\mathbb{F}_q)$, is defined by $\mathbb{F}_q^3 \setminus \{(0, 0, 0)\}$ modulo the equivalence relation $\sim$. We write $(x_0 : x_1 : x_2)$ to emphasize that the tuple belongs to $\mathbb{P}^2(\mathbb{F}_q)$ as opposed to $\mathbb{F}_q^3$. Now we can define $E(\mathbb{F}_q)$ to be the set of solutions $(X : Y : Z) \in \mathbb{P}^2(\mathbb{F}_q)$ to the curve equation $E : Y^2 = X^3 + aXZ^2 + bZ^3$. Note that we can easily map between the two representations by $(x, y) \mapsto (x : y : 1)$, $\mathcal{O} \mapsto (0 : 1 : 0)$, and $(X : Y : Z) \mapsto (X/Z, Y/Z)$ (for $Z \neq 0$), $(0 : 1 : 0) \mapsto \mathcal{O}$.

There are many ways to compute the group law on E, see [9]. 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. [31] presents addition formulas for curves in short Weierstrass form embedded in the projective plane. 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

$$\begin{aligned}
X_3 &= (X_1Y_2 + X_2Y_1)(Y_1Y_2 - a(X_1Z_2 + X_2Z_1) - 3bZ_1Z_2) \\
&\quad - (Y_1Z_2 + Y_2Z_1)(aX_1X_2 + 3b(X_1Z_2 + X_2Z_1) - a^2Z_1Z_2), \\
Y_3 &= (3X_1X_2 + aZ_1Z_2)(aX_1X_2 + 3b(X_1Z_2 + X_2Z_1) - a^2Z_1Z_2) \\
&\quad + (Y_1Y_2 + a(X_1Z_2 + X_2Z_1) + 3bZ_1Z_2)(Y_1Y_2 - a(X_1Z_2 + X_2Z_1) - 3bZ_1Z_2), \\
Z_3 &= (Y_1Z_2 + Y_2Z_1)(Y_1Y_2 + a(X_1Z_2 + X_2Z_1) + 3bZ_1Z_2) \\
&\quad + (X_1Y_2 + X_2Y_1)(3X_1X_2 + aZ_1Z_2). \qquad (1)
\end{aligned}$$

Elliptic curve cryptography [22,28] commonly relies on the hard problem called the "Elliptic Curve Discrete Logarithm Problem (ECDLP)". 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 [4,23] and countermeasures [14] have been proposed. To ensure protection against simple power analysis (SPA) attacks it is important to use regular scalar multiplication algorithms, e.g. Montgomery ladder [20] or Double-And-Add-Always [14], 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 [31]. 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 [31] 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.

[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.

Table 1. Dependencies of multiplications inside the complete addition formulas

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 2. Efficiency approximation of the number of Montgomery multipliers against the area used.

n	*Cost*	*Area* $\times$ *Time*	*Algorithm*
1	$17\mathbf{M} + 23\mathbf{a}$	$17\mathbf{M} + 23\mathbf{a}$	1 in [31]
2	$9\mathbf{M_2} + 12\mathbf{a_2}$	$18\mathbf{M} + 24\mathbf{a}$	1
3	$6\mathbf{M_3} + 8\mathbf{a_3}$	$18\mathbf{M} + 24\mathbf{a}$	2
4	$5\mathbf{M_4} + 7\mathbf{a_4}$	$20\mathbf{M} + 28\mathbf{a}$	3
5	$4\mathbf{M_5} + 6\mathbf{a_5}$	$20\mathbf{M} + 30\mathbf{a}$	4
6	$3\mathbf{M_6} + 6\mathbf{a_6}$	$18\mathbf{M} + 36\mathbf{a}$	5

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 $1 \leq n \leq 6$. We make the simple approximations 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. All algorithms and their respective Magma [11] verification code can be found in Appendices B and C. For our implementation we have chosen for $n = 3$, i. e. three Montgomery multipliers. This number of multipliers achieves a great area-time trade-off, while obtaining a good 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.

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. [26] for Microsemi® IGLOO2® FPGAs, for which the architecture is shown in Fig. 1. We give a short description of the processor in Sect. 4.1, but for more details on its internals we refer to [26]. As a consequence of building on top of this processor, we target the same FPGA. However, it is straightforward to port to other FPGA's or even ASICs which have a Montgomery multiplier with the same interface and instructions.

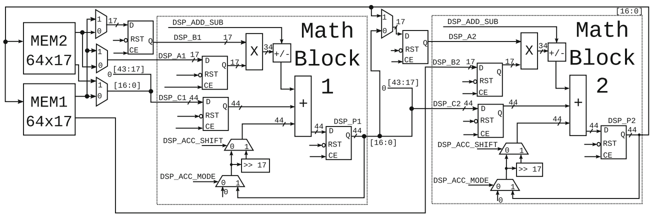

Fig. 1. Montgomery addition, subtraction and multiplication processor.

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 [14]). 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 [3]. From a very high level point of view the architecture consists of the three Montgomery multipliers and a single BRAM block, shown in Fig. 2. We note that this BRAM block is more than large enough to store the necessary temporary variables. So although Algorithm 2 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.

4.1 The Montgomery Processor

Massolino et al. [26] 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. [21]. In short, FIOS computes the partial product and partial reduction inside the same iterative loop. This can be translated into a hardware architecture, see Fig. 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 Sect. 4.4.

4.2 Memory

The main RAM memory in Fig. 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. 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 [13]. Lastly, there is a 17-bit shift register into which the scalar is loaded word by word.

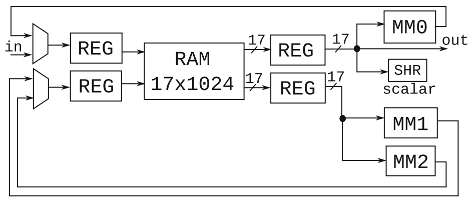

Fig. 2. Entire architecture with three Montgomery processors from [26], where MM = Montgomery processor, SHR = Shift register, REG = Register.

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 processors 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 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 [26] 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 2 there are several consecutive additions and subtractions. One example of such additions is t_9 in line 7, then latter on line 8 is added and stored on t_{10}, which on line 10 is added with a fourth value. 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 [29].

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 $P = Q$.

Line # Algorithm 2	`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	$Y_1 \leftarrow t_1 + t_4$	$X_1 \leftarrow t_9 - t_8$	
			$Z_1 \leftarrow t_{10} + t_{11}$

4.5 Scheduling

The architecture presented in Fig. 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 [26], the additions and subtractions from lines seven and eight in Algorithm 2 can be done by the otherwise idle processor `MM2` in stage six. This makes them basically free of cost.

4.6 Comparison

As our architecture supports primes from 116 to 522 bits, we can run benchmarks and do comparisons for multiple bitsizes. The results for different common prime sizes are shown in Table 5 in Appendix A. In this section we consider only the currently widely adopted 128-bit security level, presented in Table 4. Integer addition, subtraction and Montgomery modular multiplication results are the same as in Massolino et al. [26]. This is the first work implementing the new complete formulas for elliptic curves in short Weierstrass form [31], and leads to a scalar multiplication routine which takes about 14.21 ms for a 256-bit prime.

It is not straightforward to do a well-founded comparison between work in the literature. Table 4 contains different implementations of elliptic curve scalar multiplication, but they have different optimization goals. For example we top [35,36] in terms of milliseconds per scalar multiplication, but they use less multipliers or run at a lower frequency. On the other hand [1,17,18,25,27,34] 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.

A More complete results comparison

Table 4. Comparison of our results to the literature on hardware implementations for ECC. The speed results are for one scalar multiplication.

Work	Field	FPGA	Slice/ ALM	LUT	FF	Emb. Mult.	BRAM 64×18	BRAM $1\,k \times 18$	Freq. (MHz)	Scalar Mult. Cycles	(ms)
For all prime fields and prime order short Weierstrass curves											
Our	256	IGLOO 2^4	–	2828	1048	6	6	1	100	1421312	14.21
For NIST curves [29] only											
[35]	256	SmartFusion^4	–	3690	3690	0	0	12	109	2103941	19.3
[35]	256	Virtex II Pro^4	773	1546^a	1546^a	1	0	3	210	2103941	10.02
[35]	256	Virtex II Pro^4	1158	2316^a	2316^a	4	0	3	210	949951	4.52
[30]	256	Virtex 5^{6c}	1914	7656^a	7656^a	4	0	12	210	830000	3.95
[16]	192	Virtex II Pro^4	3173	6346^a	6346^a	16	0	6	93	920700^b	9.90
[32]	256	Spartan 6^6	72	193	35	8	0	24	156.25	1906250^b	12.2
[24]	256	Virtex 4^4	7020	12435	3545	8	0	4	182	993174^b	5.457
[1]	256	Virtex 6^{6c}	11.2 k	32.9 k	89.6 k^a	289	0	256	100	39922	0.40
[17]	256	Virtex 4^4	1715	2589	2028	32	0	11	490	303450	0.619
For only Edwards or Twisted Edwards curves											
[2]	192	Spartan 3E ^{4}	4654	9308^a	9308^a	0	0	0	10	125430^b	12.543
[34]	256	Zynq^{6c}	1029	2783	3592	20	0	4	200	64770	0.324
For only specific field size, but works with any prime											
[36]	256	Virtex II Pro^4	1832	3664^a	3664^a	2	0	9	108.2	3227993	29.83
[36]	256	Virtex II Pro^4	2085	4170^a	4170^a	7	0	9	68.17	1074625	15.76
[18]	256	Stratix II^4	9177	18354^a	18354^a	96	0	0	157.2	106896^b	0.68
[27]	256	Virtex II Pro^4	15755	31510^a	31510^a	256	0	0	39.46	151360	3.86
[25]	256	Virtex 4^4	4655	5740	4876	37	0	11	250	109297	0.44

^{a}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.
^{b}Values estimated by multiplying time by frequency.
^{4} ^{6} indicates LUT size.
^{c}BRAMs of Virtex 5, 6 and Zynq are $1\,k \times 36$, so they account as 2 independent $1\,k \times 18$.

Table 5. Complete comparison and results from Table 4

Work	Field	FPGA	Slice/	LUT	FF	Emb.	BRAM	BRAM	Freq.	Scalar Mult.	
			ALM			Mult.	64×18	$1\,k \times 18$	(MHz)	Cycles	(ms)
For all prime fields and prime order short Weierstrass curves											
Our	192	IGLOO 2^4	–	2828	1048	6	6	1	100	728448	7.28
Our	224	IGLOO 2^4	–	2828	1048	6	6	1	100	1036224	10.36
Our	256	IGLOO 2^4	–	2828	1048	6	6	1	100	1421312	14.21
Our	320	IGLOO 2^4	–	2828	1048	6	6	1	100	2498560	24.99
Our	384	IGLOO 2^4	–	2828	1048	6	6	1	100	3744768	37.45
Our	512	IGLOO 2^4	–	2828	1048	6	6	1	100	8187904	81.88
Our	521	IGLOO 2^4	–	2828	1048	6	6	1	100	8331832	83.32
For NIST curves [29] only											
[35]	224	SmartFusion^4	–	3690	3690	0	0	12	109	1722088	15.8
[35]	256	SmartFusion^4	–	3690	3690	0	0	12	109	2103941	19.3
[35]	224	Virtex II Pro^4	773	1546^a	1546^a	1	0	3	210	1722088	8.2
[35]	256	Virtex II Pro^4	773	1546^a	1546^a	1	0	3	210	2103941	10.02
[35]	224	Virtex II Pro^4	1158	2316^a	2316^a	4	0	3	210	765072	3.64
[35]	256	Virtex II Pro^4	1158	2316^a	2316^a	4	0	3	210	949951	4.52
[30]	256	Virtex 5^{6c}	1914	7656^a	7656^a	4	0	12	210	830000	3.95
[16]	192	Virtex II Pro^4	3173	6346^a	6346^a	16	0	6	93	920700^b	9.90
[32]	256	Spartan 6^6	72	193	35	8	0	24	156.25	1906250^b	12.2
[24]	192	Virtex 4^4	7020	12435	3545	8	0	4	182	429702^b	2.361
[24]	224	Virtex 4^4	7020	12435	3545	8	0	4	182	666666^b	3.663
[24]	256	Virtex 4^4	7020	12435	3545	8	0	4	182	993174^b	5.457
[24]	384	Virtex 4^4	7020	12435	3545	8	0	4	182	2968420^b	16.31
[24]	521	Virtex 4^4	7020	12435	3545	8	0	4	182	7048860^b	38.73
[1]	192	Virtex 6^{6c}	11.2 k	32.9 k	89.6 k^a	289	0	256	100	29948	0.30
[1]	224	Virtex 6^{6c}	11.2 k	32.9 k	89.6 k^a	289	0	256	100	34999	0.35
[1]	256	Virtex 6^{6c}	11.2 k	32.9 k	89.6 k^a	289	0	256	100	39922	0.40
[1]	384	Virtex 6^{6c}	11.2 k	32.9 k	89.6 k^a	289	0	256	100	11722	1.18
[1]	521	Virtex 6^{6c}	11.2 k	32.9 k	89.6 k^a	289	0	256	100	159959	1.60
[17]	224	Virtex 4^4	1580	1825	1892	26	0	11	487	219878	0.451
[17]	256	Virtex 4^4	1715	2589	2028	32	0	11	490	303450	0.619
For only Edwards or Twisted Edwards curves											
[2]	192	Spartan 3E ^{4}	4654	9308^a	9308^a	0	0	0	10	125430^b	12.543
[34]	256	Zynq^{6c}	1029	2783	3592	20	0	4	200	64770	0.324
For only specific field size, but works with any prime											
[36]	256	Virtex II Pro^4	1832	3664^a	3664^a	2	0	9	108.2	3227993	29.83
[36]	256	Virtex II Pro^4	2085	4170^a	4170^a	7	0	9	68.17	1074625	15.76
[18]	192	Stratix II^4	6203	12406^a	12406^a	92	0	0	160.5	70620^b	0.44
[18]	256	Stratix II^4	9177	18354^a	18354^a	96	0	0	157.2	106896^b	0.68
[18]	384	Stratix II^4	12958	25916^a	25916^a	177	0	0	150.9	203715^b	1.35
[18]	512	Stratix II^4	17017	34034^a	34034^a	244	0	0	144.97	323283^b	2.23
[27]	256	Virtex II Pro^4	15755	31510^a	31510^a	256	0	0	39.46	151360	3.86
[25]	256	Virtex 4^4	4655	5740	4876	37	0	11	250	109297	0.44

B Algorithms

Algorithm 1. 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: $(X_3 : Y_3 : Z_3) = P + Q$.

 1. $t_0 \leftarrow X_1 + Y_1;$                $t_1 \leftarrow X_2 + Y_2;$
 2. $t_2 \leftarrow Y_1 + Z_1;$                $t_3 \leftarrow Y_2 + Z_2;$
 3. $t_0 \leftarrow t_0 \cdot t_1; (\ell_3)$          $t_1 \leftarrow t_2 \cdot t_3; (\ell_5)$
 4. $t_4 \leftarrow X_1 \cdot X_2; (\ell_0)$          $t_6 \leftarrow Z_1 \cdot Z_2; (\ell_2)$
 5. $t_2 \leftarrow X_1 + Z_1;$                $t_3 \leftarrow X_2 + Z_2;$
 6. $t_0 \leftarrow t_0 - t_4;$                $t_1 \leftarrow t_1 - t_6;$
 7. $t_5 \leftarrow Y_1 \cdot Y_2; (\ell_1)$          $t_2 \leftarrow t_2 \cdot t_3; (\ell_4)$
 8. $t_7 \leftarrow a \cdot t_6; (\ell_7)$           $t_8 \leftarrow b_3 \cdot t_6; (\ell_8)$
 9. $t_9 \leftarrow t_4 - t_7;$                $t_{10} \leftarrow t_4 + t_4;$
10. $t_{11} \leftarrow t_4 + t_7;$               $t_2 \leftarrow t_2 - t_4;$
11. $t_0 \leftarrow t_0 - t_5;$                $t_1 \leftarrow t_1 - t_5;$
12. $t_2 \leftarrow t_2 - t_6;$                $t_{10} \leftarrow t_{10} + t_{11};$
13. $t_9 \leftarrow a \cdot t_9; (\ell_{10})$          $t_{11} \leftarrow b_3 \cdot t_2; (\ell_9)$
14. $t_2 \leftarrow a \cdot t_2; (\ell_8)$
15. $t_9 \leftarrow t_9 + t_{11};$               $t_8 \leftarrow t_2 + t_8;$
16. $t_6 \leftarrow t_5 - t_8;$                $t_5 \leftarrow t_5 + t_8;$
17. $t_3 \leftarrow t_1 \cdot t_9; (\ell_{12})$         $t_9 \leftarrow t_9 \cdot t_{10}; (\ell_{14})$
18. $t_{10} \leftarrow t_0 \cdot t_{10}; (\ell_{16})$       $t_0 \leftarrow t_0 \cdot t_6; (\ell_{11})$
19. $t_6 \leftarrow t_5 \cdot t_6; (\ell_{13})$         $t_1 \leftarrow t_1 \cdot t_5; (\ell_{15})$
20. $X_3 \leftarrow t_0 - t_3;$                $Y_9 \leftarrow t_6 + t_9;$
21. $Z_3 \leftarrow t_1 + t_{10};$
```

Algorithm 2. 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 3. 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 4. 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 5. 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;
```

References

1. Alrimeih, H., Rakhmatov, D.: Fast and flexible hardware support for ECC over multiple standard prime fields. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **22**(12), 2661–2674 (2014)
2. Baldwin, B., Moloney, R., Byrne, A., McGuire, G., Marnane, W.P.: A hardware analysis of twisted edwards curves for an elliptic curve cryptosystem. In: Becker, J., Woods, R., Athanas, P., Morgan, F. (eds.) ARC 2009. LNCS, vol. 5453, pp. 355–361. Springer, Heidelberg (2009). doi:10.1007/978-3-642-00641-8_41
3. Barenghi, A., Breveglieri, L., Koren, I., Naccache, D.: Fault injection attacks on cryptographic devices: theory, practice, and countermeasures. Proc. IEEE **100**(11), 3056–3076 (2012)
4. Batina, L., Chmielewski, Ł., Papachristodoulou, L., Schwabe, P., Tunstall, M.: Online template attacks. In: Meier, W., Mukhopadhyay, D. (eds.) INDOCRYPT 2014. LNCS, vol. 8885, pp. 21–36. Springer, Heidelberg (2014). doi:10.1007/978-3-319-13039-2_2
5. Bernstein, D.J.: Curve25519: new diffie-hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 207–228. Springer, Heidelberg (2006). doi:10.1007/11745853_14
6. Bernstein, D.J., Birkner, P., Joye, M., Lange, T., Peters, C.: Twisted edwards curves. In: Vaudenay, S. (ed.) AFRICACRYPT 2008. LNCS, vol. 5023, pp. 389–405. Springer, Heidelberg (2008). doi:10.1007/978-3-540-68164-9_26

7. Bernstein, D.J., Chuengsatiansup, C., Kohel, D., Lange, T.: Twisted hessian curves. In: Lauter, K., Rodríguez-Henríquez, F. (eds.) LATINCRYPT 2015. LNCS, vol. 9230, pp. 269–294. Springer, Heidelberg (2015). doi:10.1007/978-3-319-22174-8_15
8. Bernstein, D.J., Lange, T.: Faster addition and doubling on elliptic curves. In: Kurosawa, K. (ed.) ASIACRYPT 2007. LNCS, vol. 4833, pp. 29–50. Springer, Heidelberg (2007). doi:10.1007/978-3-540-76900-2_3
9. Bernstein, D.J., Lange, T.: Explicit-Formulas Database. http://hyperelliptic.org/EFD/index.html. Accessed 21 Feb 2015
10. Bernstein, D.J., Lange, T., Schwabe, P.: The security impact of a new cryptographic library. In: Hevia, A., Neven, G. (eds.) LATINCRYPT 2012. LNCS, vol. 7533, pp. 159–176. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33481-8_9
11. Bosma, W., Cannon, J.J., Playoust, C.: The Magma algebra system I: the user language. J. Symb. Comput. **24**(3/4), 235–265 (1997)
12. Certicom Research. SEC 2: Recommended Elliptic Curve Domain Parameters, Version 2.0. Technical report, Certicom Research (2010)
13. Clavier, C., Joye, M.: Universal exponentiation algorithm a first step towards *Provable* SPA-Resistance. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 300–308. Springer, Heidelberg (2001). doi:10.1007/3-540-44709-1_25
14. Coron, J.-S.: Resistance against differential power analysis for elliptic curve cryptosystems. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 292–302. Springer, Heidelberg (1999). doi:10.1007/3-540-48059-5_25
15. ECC Brainpool: ECC Brainpool standard curves and curve generation. Technical report, Brainpool (2005)
16. Fan, J., Sakiyama, K., Verbauwhede, I.: Elliptic curve cryptography on embedded multicore systems. Design Autom. Embedded Syst. **12**(3), 231–242 (2008). doi:10.1007/s10617-008-9021-3
17. Güneysu, T., Paar, C.: Ultra high performance ECC over NIST primes on commercial FPGAs. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 62–78. Springer, Heidelberg (2008). doi:10.1007/978-3-540-85053-3_5
18. Guillermin, N.: A high speed coprocessor for elliptic curve scalar multiplications over $\mathbb{F}_p$. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 48–64. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15031-9_4
19. Hamburg, M.: Ed448-Goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625 (2015). http://eprint.iacr.org/2015/625.pdf
20. Joye, M., Yen, S.-M.: The montgomery powering ladder. In: Kaliski, B.S., Koç, K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 291–302. Springer, Heidelberg (2003). doi:10.1007/3-540-36400-5_22
21. Koç, Ç.K., Acar, T., Kaliski, B.S.: Analyzing and comparing Montgomery multiplication algorithms. IEEE Micro **16**(3), 26–33 (1996)
22. Koblitz, N.: Elliptic curve cryptosystems. Math. Comput. **48**, 203–209 (1987)
23. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999). doi:10.1007/3-540-48405-1_25
24. Loi, K.C.C., Ko, S.B.: Scalable elliptic curve cryptosystem FPGA processor for NIST prime curves. IEEE Trans. Very Large Scale Integration (VLSI) Syst. **23**(11), 2753–2756 (2015)
25. Ma, Y., Liu, Z., Pan, W., Jing, J.: A high-speed elliptic curve cryptographic processor for generic curves over GF*(p)*. In: Lange, T., Lauter, K., Lisoněk, P. (eds.) SAC 2013. LNCS, vol. 8282, pp. 421–437. Springer, Heidelberg (2014)

26. Massolino, P.M.C., Batina, L., Chaves, R., Mentens, N.: Low Power Montgomery Modular Multiplication on Reconfigurable Systems. Cryptology ePrint Archive, Report 2016/280 (2016). http://eprint.iacr.org/2016/280
27. McIvor, C., McLoone, M., McCanny, J.V.: Hardware elliptic curve cryptographic processor over GF(p). IEEE Trans. Circuits Syst. I Regul. Pap. **53**(9), 1946–1957 (2006)
28. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986). doi:10.1007/3-540-39799-X_31
29. National Institute for Standards and Technology. Federal information processing standards publication 186–4. digital signature standard. Technical report, NIST (2013)
30. Pöpper, C., Mischke, O., Güneysu, T.: MicroACP - a fast and secure reconfigurable asymmetric crypto-processor. In: Goehringer, D., Santambrogio, M.D., Cardoso, J.M.P., Bertels, K. (eds.) ARC 2014. LNCS, vol. 8405, pp. 240–247. Springer, Heidelberg (2014). doi:10.1007/978-3-319-05960-0_24
31. Renes, J., Costello, C., Batina, L.: Complete addition formulas for prime order elliptic curves. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9665, pp. 403–428. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49890-3_16
32. Roy, D.B., Das, P., Mukhopadhyay, D.: ECC on your fingertips: a single instruction approach for lightweight ECC design in GF(p). In: Dunkelman, O., Keliher, L. (eds.) SAC 2015. LNCS, vol. 9566, pp. 161–177. Springer, Heidelberg (2015)
33. Sakiyama, K., Batina, L., Preneel, B., Verbauwhede, I.: Superscalar coprocessor for high-speed curve-based cryptography. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 415–429. Springer, Heidelberg (2006). doi:10.1007/11894063_33
34. Sasdrich, P., Güneysu, T.: Efficient elliptic-curve cryptography using curve25519 on reconfigurable devices. In: Goehringer, D., Santambrogio, M.D., Cardoso, J.M.P., Bertels, K. (eds.) ARC 2014. LNCS, vol. 8405, pp. 25–36. Springer, Heidelberg (2014). doi:10.1007/978-3-319-05960-0_3
35. Varchola, M., Guneysu, T., Mischke, O.: MicroECC: A lightweight reconfigurable elliptic curve crypto-processor. In: 2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig), pp. 204–210, November 2011
36. Vliegen, J., Mentens, N., Genoe, J., Braeken, A., Kubera, S., Touhafi, A., Verbauwhede, I.: A compact FPGA-based architecture for elliptic curve cryptography over prime fields. In: 2010 21st IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP), pp. 313–316, July 2010
37. Yao, G.X., Fan, J., Cheung, R.C.C., Verbauwhede, I.: Faster pairing coprocessor architecture. In: Abdalla, M., Lange, T. (eds.) Pairing 2012. LNCS, vol. 7708, pp. 160–176. Springer, Heidelberg (2013). doi:10.1007/978-3-642-36334-4_10
38. Yen, S., Joye, M.: Checking before output may not be enough against fault-based cryptanalysis. IEEE Trans. Comput. **49**(9), 967–970 (2000)