Multi-core FPGA Implementation of ECC with Homogeneous Co-Z Coordinate Representation

Bo-Yuan Peng[1], Yuan-Che Hsu[2], Yu-Jia Chen[2], Di-Chia Chueh[2], Chen-Mou Cheng[3], and Bo-Yin Yang[1]

[1] Academia Sinica, Taiwan
{bypeng,by}@crypto.tw
[2] National Taiwan University, Taiwan
{b01901138,b01901017,b01901020}@ntu.edu.tw
[3] Osaka University, Japan
chenmou.cheng@gmail.com

Abstract. Elliptic Curve Cryptography (ECC) is gaining popularity in recent years. Having short keys and short signatures in particular makes ECC likely to be adopted in numerous internet-of-things (IoT) devices. It is therefore critical to optimize ECC well for both speed and power consumption. Optimization opportunities exist on several different levels: algorithm, architecture, and/or implementation. We combine optimizations at every level in an efficient multi-core FPGA implementation. The core building block for our implementation is a Montgomery multiplier capable of modular additions and multiplications with an arbitrary prime modulus. The size of the prime modulus can also be changed easily, for which we have implemented and tested up to 528-bits used in the NIST P-521 curve. Based on this building block, we have developed a multi-core architecture that supports multiple parallel modular additions, multiplications, and inverses. Efficient ECC group addition and doubling are then built from this foundation. To support a wide variety of curves and at the same time resist timing/power-based side-channel attacks, our scalar multiplication is implemented using the Co-Z ladder due to Hutter, Joye, and Sierra. This approach also allows us to trade off between speed and power consumption by using a different number of Montgomery cores.

Keywords: ECC, Co-Z, Multi-Core, FPGA, Montgomery Reduction.

1 Introduction

Elliptic Curve Cryptography (ECC) was invented independently by Koblitz and Miller [1, 2] in the 1980s and has seen increasing use for information security in the last decade. The most important operations in ECC are scalar multiplication and point (group) addition. Most ECC implementations require many arithmetic operations modulo a prime (of 256–521 bits, as per the desired security level). These are complex, resource-intensive operations.

There are many different techniques to speed up ECC operations, e.g.:

1. Choose a curve of a special form, such as Montgomery or Twisted Edwards Curves [3, 4] where the number of modular multiplications can be reduced.
2. Choose a better representation for the point on the curve. Every extant representation uses a scaling factor to avoid computing expensive modular inverses. On short Weierstrass curves, the most common representations are homogeneous and Jacobian projective coordinates.
3. Better big-integer modular arithmetic such as Montgomery Reductions [5].

As more and more devices are connected into the "internet of things", efficient network security solutions become necessaries with many possible trade-offs among cost, power consumption, security level, and flexibility.

Flexibility in the security solution can be achieved via FPGAs, a practice that recently became more fashionable [6]. FPGAs are reprogrammable and any excess LUTs can always be used for extra functionality. In networking applications, a functional unit may be cloned dozens of times on the FPGA and properly scheduled to run many similar operations simultaneously. We can do the same with the big integer modular multiplications in ECC. An obvious caveat is that n copies of the key unit does not automatically make things n times as fast, as there may be bottlenecks scheduling the critical operations.

In this work, we have developed a computing architecture with multiple Montgomery Reduction cores (from 2 to many). Each Montgomery Reduction core includes two multipliers and completes one 528-bit Montgomery multiplication in 66 cycles with frequency faster than 30 MHz on the Xilinx® Zynq-7000™ All Programmable SoC. We use this architecture to implement a general high-security ECC engine compatible with all short Weierstrass Curves and prime moduli up to 521 bits. These include the 256-bit secure NIST [15] and Brainpool [16] curves.

Because we cannot restrict ourselves to Montgomery or Edwards curves, we use the Co-Z ladder by Hutter *et al.* [8] for scalar multiplications (and Cohen *et al.* [9] for point additions), which is resilient against power or timing attacks.

Our design is modular and scalable (in numbers of cores and also bitlength down to 256); we also describe, to the best of our knowledge, for the first time how the Co-Z ladder can be flexibly implemented with Montgomery reduction units. The main ladderstep can be performed in as few as 3 rounds of big integer multiplications (plus extra modular additions) with 12 Montgomery cores. In addition, we provide good scheduling for 1, 2, 3 or 5 Montgomery cores.

The rest of this paper is structured as follows:

- Section 2 surveys the history and related works, including the formulas that evaluates the scalar multiplications and designs related to what we used.
- Section 3 analyzes the degree of parallelism of the Co-Z ladderstep and describe the requirements an optimal scheduling of this step.
- Section 4 describes our hardware architecture with modular and flexible Montgomery multipliers and how we arrived at the design.
- Section 5 describes our implementation and test results in more detail.
- Section 6 is the Conclusion and discusses possible future work.

2 History and Related Work

ECC and Notations. Koblitz and Miller [1, 2] independently suggested using discrete logarithms on the rational points of an elliptic curve group over a finite field for cryptosystems. In this paper we stick to nonsingular (a, $b \in \mathbb{F}_p$ with $4a^3 + 27b^2 \neq 0$) elliptic curves in short Weierstrass form over prime fields,

$$\mathcal{E}(\mathbb{F}_p) := \{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p |\, y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\} \tag{1}$$

$\mathcal{E}(\mathbb{F}_p)$ comprises points satisfying the curve equation $\mathcal{E} : y^2 = x^3 + ax + b$ plus a "point at infinity" $\mathcal{O}$. We can define an abelian group on $\mathcal{E}(\mathbb{F}_p)$ such that $\mathcal{O}$ is the unit element, and any three co-linear points add up to $\mathcal{O}$.

The group order is denoted ℓ (with $\ell/p \approx 1$), which we want to be a small multiple of a large prime q. The *scalar multiplication* $\mathbf{Q} = \langle k \rangle \mathbf{P}$, is defined as repeated addition on k copies of a point $\mathbf{P}$. On good curves, it is difficult given $\mathbf{P}$ and $\mathbf{Q}$ to find k such that $\mathbf{Q} = \langle k \rangle \mathbf{P}$ (takes time $\Theta(\sqrt{q})$ with the best methods we know). This is the elliptic curve discrete logarithm problem (ECDLP). A scalar multiplication $\mathbf{Q} = \langle k \rangle \mathbf{P}$ in contrast takes time polylog(p) given k and $\mathbf{P}$.

Computing Scalar Multiplication and Coordinates. In the early days of cryptography, exponentiation in groups in general including scalar multiplications on elliptic curve groups are performed using the double-and-add approach. However, Kocher noted that we can break such implementations using *side-channel attacks* [10], by observing timing or power usage patterns. Even "double-and-always-add" approaches [11] designed to evade Kocher's attacks at the cost of time and energy can be vulnerable to other, more advanced side-channel attacks [12, 13].

An approach to scalar multiplication generally resilient to side-channel attacks is *differential addition chains.* The original example, where we can compute $\langle 2 \rangle \mathbf{Q}$ and $\mathbf{P} + \mathbf{Q}$ from $\mathbf{P}, \mathbf{Q}, \mathbf{P} - \mathbf{Q}$ (Alg. 1), is the *Montgomery ladder* [4].

Algorithm 1: Montgomery Ladder [4]

```
  Input: $\mathbf{P} \in \mathcal{E}(\mathbb{F}_p)$, $k = (k_{n-1}, ..., k_0)_2 \in \mathbb{N}$
  Output: $\mathbf{Q} = \langle k \rangle \mathbf{P}$
1 $\mathbf{R_0} = \mathcal{O}$; $\mathbf{R_1} = \mathbf{P}$
2 for $i = n - 1$ downto $0$ do
3     $t \leftarrow k_i$
4     $\mathbf{R_{1-t}} \leftarrow \mathbf{R_{1-t}} + \mathbf{R_t}$
5     $\mathbf{R_t} \leftarrow \langle 2 \rangle \mathbf{R_t}$
6 end
7 return $\mathbf{R_0}$
```

Curve Form and Point Representations. [4] uses two numbers in $\mathbb{F}_p$ to represent each of two points $\mathbf{R_0}, \mathbf{R_1}$ on the curve. That would be normal, except the two numbers for each point only relate directly to one coordinate ($x_0 = X_0/Z_0$, $x_1 = X_1/Z_1$) out of two. The *Montgomery* curve $y^2 = x^3+4Ax^2+x$ is used to facilitate the Montgomery Ladder. These choices of curve, algorithm and representation are synergetic, being efficient and naturally resilient against side channels.

The representation $x = X/Z$, a *homogeneous projective coordinate*, demonstrates the almost universal use of *scaling factors* to avoid computating inverses in $\mathbb{F}_p$. In fact, for short Weierstrass curves alone, there are two different common sets of projective coordinates: *homogeneous*, where (X, Y, Z) denotes the point $(X/Z, Y/Z)$; and *Jacobian*, where $(X : Y : Z)$ denotes the point $(X/Z^2, Y/Z^3)$.

The Montgomery-curve-and-ladder combination is still one of the best methods to implement ECC for security and speed today, illustrating the importance of a good set of choices of algorithm, curve and representation (cf. [14]).

Co-Z ladder for NIST Curves. Most of the time twisted Edwards curves [3] (all birationally equivalent to Montgomery curves) offers the best all-around performance. However, in some cases compatibility for short Weierstrass curves not equivalent to Montgomery curves (e.g., NIST [15] /Brainpool [16] curves) is required.

In 2011, Hutter *et al.* described what as far as we know is the best general differential chain implementation for scalar multiplications on a general short Weierstrass curve [8]. Like the original Montgomery chain, only the X-parts of homogeneous projective coordinates need to be kept. One of the main features of the new method is that the two points $\mathbf{R_0}$ and $\mathbf{R_1}$ shares Z coordinates during each ladderstep. The y part of the result can be reconstructed at the end, enabling compressed public keys and signatures. We call this the *Co-Z ladder.*

Co-Z Ladder Formulas. Let $\mathbf{P_1} = (X_1, Y_1, Z)$, $\mathbf{P_2} = (X_2, Y_2, Z)$ and $\mathbf{P_1} - \mathbf{P_2} = \pm\mathbf{P}$ where $\mathbf{P} = (x_P, y_P)$. Further let $\mathbf{P_1} + \mathbf{P_2} = (X_1', Y_1', Z')$ and $\langle 2\rangle\mathbf{P_2} = (X_2', Y_2', Z')$. Given $(X_1,\ X_2,\ Z)$ and x_P we can compute $(X_1',\ X_2',\ Z')$ via

$$\begin{cases} X_1' = V[(X_1+X_2)(X_1^2+X_2^2-U+2aZ^2)+4bZ^3-x_PZU] \\ X_2' = U[(X_2^2-aZ^2)^2-8bZ^3X_2] \\ Z' = UVZ \end{cases} \tag{2}$$

where $U = (X_1 - X_2)^2$ and $V = 4X_2(X_2^2 + aZ^2) + 4bZ^3$. In the appendix, we give the complete process (Alg. 3) to evaluate formula (2). The effort in the computation is essentially 11 regular big-integer multiplications (denoted as $\mathfrak{M}$) and 5 big-integer squarings (denoted as $\mathfrak{S}$), or $11\mathfrak{M} + 5\mathfrak{S}$. Note that some multiplications are by a and $4b$, which may not be as large.

Hutter *et al.* also noted we can forget Z and instead track $(X_1,\ X_2,\ T_P,\ T_a,\ T_b)$ where $T_P = x_PZ$, $T_a = aZ^2$ and $T_b = 4bZ^3$, with $10\mathfrak{M}+5\mathfrak{S}$ per ladderstep with the formulas (3). The appendix details the complete procedure in Alg. 4.

$$\begin{cases} T_P' = T_P W \\ T_a' = T_a W^2 \\ T_b' = T_b W^3 \\ X_1' = V[(X_1 + X_2)(X_1^2 + X_2^2 - U + 2T_a) + T_b] - T_P' \\ X_2' = U[(X_2^2 - T_a)^2 - 8X_2 T_b] \end{cases} \tag{3}$$

where $U = (X_1 - X_2)^2$, $V = 4X_2(X_2^2 + T_a) + T_b$ and $W = UV$.

Recovering y Coordinates. The Co-Z ladder usually ignores the y coordinates of the elements in $\mathcal{E}(\mathbb{F}_p)$ However, in some protocols (such as MQV) the y coordinate of $\mathbf{Q} = \langle k \rangle \mathbf{P}$ is required. If using differential addition of Alg. 3, corresponding to formulas (2), we see the output (X_1, X_2, Z), We can compute $\mathbf{Q} = (X_1', Y_1', Z')$ using $10\mathfrak{M} + 2\mathfrak{S}$ as follows (Alg. 5):

$$\begin{cases} X_1' = 4y_P X_1 Z^2 \\ Y_1' = 2[(x_P X_1 Z + aZ^2)(x_P Z + X_1) - X_2(x_P Z - X_1)^2] + 4bZ^3 \\ Z' = 4y_P Z^3 \end{cases} \tag{4}$$

If using Alg. 4, corresponding to formulas (3), we see the output (X_1, X_2, Z), We can compute $\mathbf{Q} = (X_1', Y_1', Z')$ using $10\mathfrak{M} + 3\mathfrak{S}$ as follows (Alg. 6):

$$\begin{cases} X_1' = 4y_P x_P T_P^2 X_1 \\ Y_1' = x_P^3[T_b + 2(T_P X_1 + T_a)(X_1 + T_P) - 2X_2(X_1 - T_P)^2] \\ Z' = 4y_P T_P^3 \end{cases} \tag{5}$$

This Co-Z ladder from Hutter *et al.* is not listed on [14], which is usually a very comprehensive reference. We believe that it is currently the best scalar multiplication method on general short Weierstrass curves.

Components of the Co-Z Ladder. A scalar multiplication in the NIST P-521 curve takes about 8000 modular multiplications (treating squaring and multiplication as the same), and we are yet to build the requisite multiplier. The multi-staged Montgomery reduction method [5] is presently the de facto standard approach for generic modular multiplications. This is well-studied and we omit details about Montgomery modular multiplier unit. In practice our implementation need to fit the FPGA platform and the practical requirement.

3 Task Scheduling in the Co-Z Ladder

Whenever multipliers are being added, two questions are inevitably raised:

1. Can n Montgomery cores be used efficiently? I.e. can Algorithm 3–6 be completed within $\lceil 16/n \rceil$, $\lceil 15/n \rceil$, $\lceil 12/n \rceil$, and $\lceil 13/n \rceil$ rounds of big-integer multiplications, respectively? (Of course, Alg.3 and 4 are more important.)

2. What is the least number of rounds of big-integer multiplications for Alg. 3–4 (and less importantly Alg. 5 and 6)? How many Montgomery cores are required in each case?

These are classical problems in parallel computing, that becomes practical to those seeking to speed up ECC. To solve this problem, the targeted algorithm will be transformed into a *task schedule graph*, which is a directed acyclic graph (DAG) in which each directed edge implies the causality between the steps (vertices in the graph) in the algorithm. Solving the famous DAG scheduling problem then gives the answers to the above two questions [18].

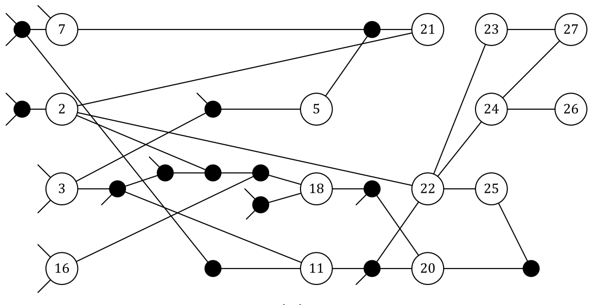
Here we analyze the degrees of parallelism of the more significant Algorithm 3–4. See the appendix for the corresponding Algorithm 5–6. Compared with multiplication, the big-integer addition/subtraction is much faster, so we will focus on the scheduling of former. Preferably, we would like to have all our Montgomery cores to run and stop (almost) at the same time to ease our scheduling task. The relationship between the number of Montgomery cores and the number of rounds of big-integer multiplications required is shown in Table 1.

Table 1. The number of rounds of big-integer multiplications required to perform Co-Z ladder in [8]. In daggered (†) cases, small and fixed a and b may improve the performance by one round.

# cores	Alg. 3 ($11\mathfrak{M} + 5\mathfrak{S}$)	Alg. 4 ($10\mathfrak{M} + 5\mathfrak{S}$)	Alg. 5 ($10\mathfrak{M} + 2\mathfrak{S}$)	Alg. 6 ($10\mathfrak{M} + 3\mathfrak{S}$)
2	8 rounds†	8 rounds	6 rounds†	7 rounds
3	6 rounds†	5 rounds	4 rounds	5 rounds
4	5 rounds	5 rounds	4 rounds	4 rounds
5	4 rounds	5 rounds	3 rounds	3 rounds
enough	3 rounds/12 cores	4 rounds/? cores	3 rounds/5 cores	3 rounds/5 cores

3.1 Critical Paths Evaluating $(X_1, X_2, T_P, T_a, T_b)$

We start with Algorithm 4 and 6 as fewer times of big-integer multiplications are required in Algorithm 4 than in Algorithm 3, and the differential addition will be performed for a lot of times in Montgomery ladder. To find the critical paths of the algorithms, we need to transform them into task schedule graphs. The task schedule graph of algorithm 4 is shown in Figure 1(a). For all of the task schedule graphs in this paper, each edge indicates the causality between the connected blocks (vertices), where the left block is performed earlier than the right one. The floating edges imply that it is necessary to retrieve the input data of the algorithm to perform the connected blocks. The white blocks indicate the big-integer multiplications, and the black ones indicate the big-integer additions or subtractions. We can omit all of the black ADD/SUB blocks in order to find the lengths of the critical paths with respect to big-integer multiplication blocks. The graph with ADD/SUB omitted is given in Figure 1(b).



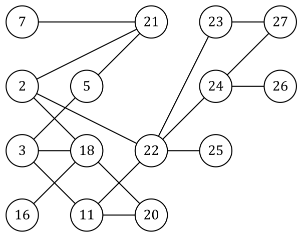


Fig. 1. The task schedule graph of Algorithm 4. The full schedule is shown in (a) and ADD/SUB blocks are omitted in (b).

Observing in Figure 1(b), we can easily find the critical paths of Algorithm 4 with respect to big-integer multiplications are $\langle 3 \to 11 \to 22 \to \{23, 24\} \to \{26, 27\}\rangle$ ($23 \to 26$ invalid), and the length is 5. Recalling that $10\mathfrak{M} + 5\mathfrak{S}$ are required (15 in total), if Algorithm 4 is adopted to perform formula (3), it may be efficient (that is, without any core always idle) to construct a 2-Montgomery-core and a 3-Montgomery-core schedule, but a 4-Montgomery-core schedule cannot be efficient if a 3-Montgomery-core schedule is found.

3.2 2 and 3 Cores

To construct the 2-core and 3-core schedules for Algorithm 4 we will follow the strategies below.

1. One round of big-integer multiplication on each core will start at the same time. We can use extra caution in implementing the Montgomery reductions such that all the cores stop at the same time.
2. If there are different schedules that run with the same number of rounds of big-integer multiplications, pick the schedule using fewer big-integer registers. If all of the candidate schedules use the same number of large number registers, pick the schedule with fewer rounds of ADD/SUB operations.

The 2-core and 3-core schedules for Algorithm 4 are shown as Algorithm 7 and 8, respectively. We can see that the 2-core schedule takes 8 rounds, while the 3-core, 5 rounds.

3.3 4 and 5 Cores: Critical Paths Evaluating (X_1, X_2, Z)

We have shown that the lengths of critical paths in Algorithm 4 is 5 with respect to big-integer multiplication rounds, which means that it is not efficient to construct a 4-Montgomery-core system to perform Algorithm 4. Algorithm 4 is designed to reduce the rounds of big-integer multiplications in a single-Montgomery-core system, as Algorithm 3 requires one more multiplication. In multi-core systems, the main concern is not the total number of multiplications performed but the time to finish running all the multiplications. It may be possible to build a schedule for Algorithm 3 with shorter critical path length. The

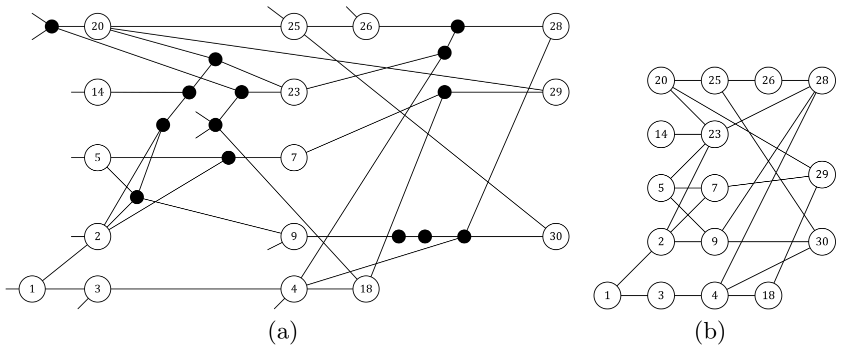


Fig. 2. The task schedule graph of Algorithm 3.

task schedule graph for Algorithm 3 is given in Figure 2. The critical path is $\langle 1 \rightarrow 3 \rightarrow 4 \rightarrow 18 \rightarrow 28 \rangle$, but actually this critical path can be reduced. $4bZ^3$ is evaluated at step 4 in Algorithm 3, and here we shall inspect in more detail all of the predecessor steps related to step 4.

$R_1 \leftarrow Z^2$	$R_1 \leftarrow Z^2$
$R_3 \leftarrow Z \times R_1$	$R'_3 \leftarrow 4b \times Z$
$R_4 \leftarrow 4b \times R_3$	$R'_4 \leftarrow R_1 \times R'_3$

We can see $4bZ^3$ is originally evaluated as the product of $4b$ and Z^3, but actually it can be evaluated from the product of $4bZ$ and Z^2, both of which are of degree 2. Since Z^3 is referenced only in step 4, we can modify step 3 and 4 in Algorithm 3, resulting in the modified task schedule graph shown in Figure 3. Now the critical path length is 4 with respect to rounds of big-integer multiplications. This observation may be generalized to more cases. To evaluate the value of a monomial $x = x_0^{r_0}, ..., x_{n-1}^{r_{n-1}}$ with total degree $r = r_0 + ... + r_{n-1}$, it is the best to generate the divisor $x_a = x_0^{a_0}, ..., x_{n-1}^{a_{n-1}}$ and $x_b = x_0^{b_0}, ..., x_{n-1}^{a_{n-1}}$, where $x = x_a x_b$, $\lceil r/2 \rceil - 1 \leq a = a_0 + ... + a_{n-1} \leq \lceil r/2 \rceil$ and $\lceil r/2 \rceil - 1 \leq b = b_0 + ... + b_{n-1} \leq \lceil r/2 \rceil$ if we want to shrink the critical path generating x. This observation will be important in subsection 3.4.

A 5-Montgomery-core schedule with 4 rounds can be created directly from Figure 3(b). Observing that $11\mathfrak{M} + 5\mathfrak{S}$ are required in the algorithm, as well as that only step 28, 29, and 30 (3 in total) can be performed in the last round, there will be 13 big-integer multiplications yet to be performed *before* the last round. Therefore, there exists a 4-Montgomery-core system that can perform Algorithm 3 in 5 rounds of big-integer multiplications. An additional note is

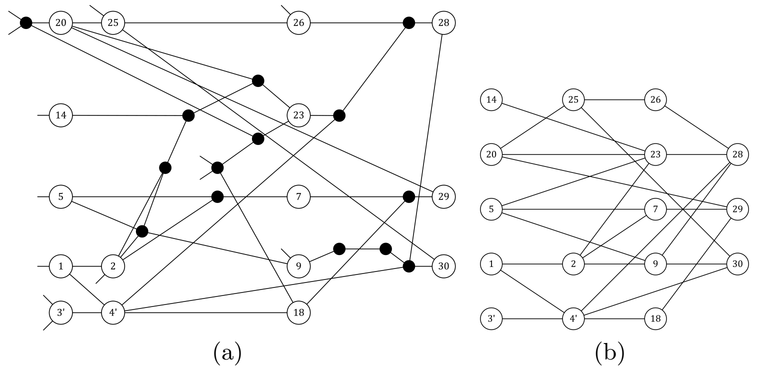


Fig. 3. The task schedule graph with step 3 and 4 modified in Algorithm 3.

that when it is the case that a and $4b$ are constants, both 2-Montgomery-core and 3-Montgomery-core systems can perform Algorithm 3 with one round fewer than original cases. This fact makes it more competitive than Algorithm 4.

3.4 How Many Cores Are Required for the Fastest Performance?

The next problem in implementing the Co-Z approach is the resource requirement if we want to speed up the evaluations to the extreme. How many rounds of big-integer multiplications are required at least? In this case, how many cores are required?

By induction, we can see that to evaluate a monomial of degree n, the best approach will take $\lceil \lg n \rceil$ rounds of multiplications. To estimate the requirement, let us start with the degrees of the elements in formula (4). We can easily see $\deg(X_1') = \deg(Z') = 4$ with respect to (X_1, X_2, Z, x_P, y_P, a, $4b$) and then at least 2 rounds of big-integer multiplications are required to evaluate both X_1' and Z'. Observing that the elements of the highest degree in Y_1' are of degree 5, at least 3 rounds of big-integer multiplications are required to evaluate Y_1'. Similar reasoning applies to formula (5), and 3 rounds of big-integer multiplications are necessary. Therefore, a 5-Montgomery-core system can lead to the best performance of time to recover (X_1', Y_1', Z').

Similar analysis can be applied to formula (2), but the story is more complicated. It is easy to see the degrees of U, V, X_1', X_2' and Z' in formula (2) with respect to (X_1, X_2, Z, x_P, a, $4b$) are 2, 4, 8, 8 and 7, respectively. It is obvious that evaluating U using one big-integer multiplication is optimal. It is a good news that $X_1' = V \times f_{X_1'}(X_1, X_2, Z, x_P, a, 4b)$ where both V and $f_{X_1'}(\cdot)$ are of degree 4, which means we may optimize the evaluation procedure of V and $f_{X_1'}()$ (with 2 rounds of big-integer multiplications) and then get the optimal flow to evaluate X_1'. The fact $Z' = UVZ = UZ \times V$ makes the optimization procedure of evaluating Z' to depend also on that of evaluating V.

$X_2' = U \times f_{X_2'}(X_1, X_2, Z, x_P, a, 4b)$ brings a problem, as $f_{X_2'}(\cdot)$ is of degree 6. A better approach is to try to factor $f_{X_2'}(\cdot)$ as a product of a quadratic polynomial and a quartic polynomial, but it is impossible here. A second choice is given by

$$\begin{aligned} f_{X_2'} &= (X_2^2 - aZ^2)^2 - 8bZ^3X_2 \\ &= X_2^4 - 2aX_2^2Z^2 + a^2Z^4 - 8bZ^3X_2 \\ &= X_2^2(X_2^2 - 2aZ \times Z) + Z^2[(aZ)^2 - 8bX_2Z] \end{aligned} \tag{6}$$

We can evalaute UX_2^2, UZ^2, $X_2^2 - 2aZ^2$, and $(aZ)^2 - 8bX_2Z$ in 2 rounds of big-integer multiplication and then find X_2' in the next round (3 rounds in total). To see how to evaluate V and $f_{X_1'}(\cdot)$, we observe that

$$\begin{aligned} V &= 4X_2(X_2^2 + aZ^2) + 4bZ^3 \\ &= 4X_2 \times X_2^2 + 4X_2Z \times aZ + 4bZ \times Z^2 \end{aligned} \tag{7}$$

$$\begin{aligned} f_{X_1'} &= 2(X_1 + X_2)(X_1X_2 + aZ^2) + 4bZ^3 - x_PZU \\ &= 2X_1X_2(X_1 + X_2) + aZ \times 2Z(X_1 + X_2) + 4bZZ^2 - x_PZU \end{aligned} \tag{8}$$

Now we can build a 12-Montgomery-core system to perform Algorithm 3, and the key schedule how to use the Montgomery cores is given as follows:

Table 2. The key schedule to perform Algorithm 3.

Round	List of Multiplication
1	U, X_2^2, aZ, Z^2, X_2Z, $4bZ$, x_PZ, $(X_1 + X_2)Z$, X_1X_2
2	$M_1 = U \cdot X_2^2$, $M_2 = aZ \cdot Z$, $M_3 = U \cdot Z^2$, $M_4 = (aZ)^2$, $M_5 = 4b \cdot X_2Z$, $M_6 = X_2 \cdot X_2^2$, $M_7 = aZ \cdot X_2Z$, $M_8 = 4bZ \cdot Z^2$, $M_9 = U \cdot Z$, $M_{10} = (X_1 + X_2)Z \cdot aZ$, $M_{11} = (X_1 + X_2)X_1X_2$, $M_{12} = U \cdot x_PZ$ $f_{X_1'} = 2M_{11} + 2M_{10} + M_8 - M_{12}$ $V = 4M_6 + 4M_7 + M_8$
3	$X_1' = N_1 = V \cdot f_{X_1'}$, $Z' = N_2 = M_9 \cdot V$ $N_3 = M_1 \cdot (X_2^2 - 2M_2)$, $N_4 = M_3 \cdot (M_4 - 2M_5)$, $X_2' = N_3 + N_4$

We give only one remark here for formula (3). With respect to (X_1, X_2, T_P, T_a, T_b), the degrees of T_a' and T_b' are 11 and 16, respectively. To perform Algorithm 4, at least 4 rounds of big-integer multiplications will be necessary. Since we have already found a 5-Montgomery-core system that perform one differential addition in 4 rounds of multiplications, it is not suggested to implement a system that performs Algorithm 4 in 4 rounds of big-integer multiplications.

4 Multi-Montgomery-Core Hardware Architecture

In this section, we suggest a hardware architecture with multiple Montgomery reduction cores. A typical Montgomery reduction core evaluates

$$R = \mathbf{MM}_D(A, B, P) = ABD^{-n} \mod P \tag{9}$$

where A, B, P are of n digits at most in D-ary presentation. We note that P is usually the prime number p forming the prime field $\mathbb{F}_p$ or the group order of the elliptic curve group $q = |\mathcal{E}|$, which is fixed during one operation of ECC. This suggests that $P^{-1} \mod D$ can be pre-computed during the setup phase, so we will assume that it is ready for us. Therefore, each Montgomery multiplier will get two inputs (A, B) and generate one output R. A classical 2R1W block memory can provides the bus width enough for one Montgomery multiplier. When there are 2 or more Montgomery multipliers, a typical choice to use a MUX/deMUX to collect the outputs of the Montgomery multipliers, and to dispatch the value in the memory to the specified inputs of the multipliers. This approach will cost more cycles on the MUX/deMUX.

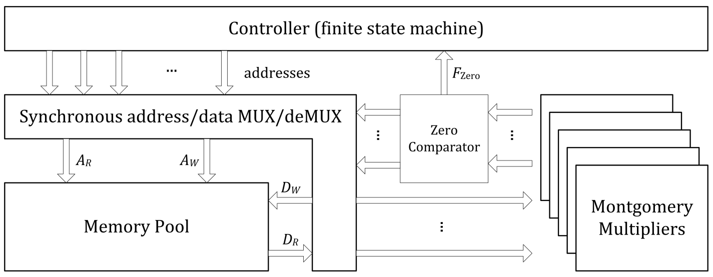


Fig. 4. The proposed block diagram.

Figure 4 illustrates the hardware architecture of a multi-multiplication-core system. A finite state machine or a controller handles the addresses for the memory pool. There are paths from the input of the whole system to the write-data buses, and the controller can assign some pre-defined direct values to the write-data buses. The data buses do not bother the controller directly, but there are some cases in which we need to check if the outputs of the large number arithmetic units become 0. For example, we need to check if both the input [23] and the output (defending the fault attacks) affine (x, y)-coordinates implies valid elements in the elliptic curve group. One comparator to zero, whose comparison result is a flag for the controller, is installed from the output bus of each large number arithmetic unit.

5 Implementation and Results

We show our result with a 3-Montgomery-core system, a 5-Montgomery-core system, and/or a 12-Montgomery-core system. For the 3-core and the 5-core system, the maximum bit sizes are scalable and we provide the results for 264-bit (suitable for 256-bit fields) and for 528-bit (suitable for 521-bit or 512-bit fields) operations in ECC. The Montgomery reduction cores, standing for the big integer multipliers, are of base $d = 2^8$. The detailed design for the the

Montgomery reduction cores is shown in the appendix. A remark is given here that additional BRAM blocks (named as $xP^{-1}P$ pools) are allocated in order to restore pre-evaluated values that are often used in the Montgomery cores.

In this work, Xilinx® Zynq-7000™ All Programmable SoC (APSoC) on AVNet® / Digilent® ZedBoard™ is adopted for the 3-core system, and that on Xilinx® ZC706 Evaluation Kit is adopted for the 5-core system and the 12-core system. We also show our result about the resource requirement for multiple 3-core and 5-core ECC engines in one system on ZC706 board, which implies that to build multiple 3-core engines in one system is better if there are sufficient resources to build a 12-core system.

No DSP slices are used in the system as DSP slices may be used for multimedia purpose for other components implemented in it.

In our system, the functions of ECC operations that are often used include:

1. Re-configurable parameters of a, b, p, $q = |\mathcal{E}|$, and the base point G.
2. Scalar multiplication with the scalar k and the element $\mathbf{P}$ in the elliptic group $\mathcal{E}$.
3. Group point addition of the elements $\mathbf{P}$ and $\mathbf{Q}$ in the elliptic group $\mathcal{E}$. The classical approach by Cohen *et al.* [9] is applied.
4. Large number multiplication, addition, and subtraction modulo the group order $q = |\mathcal{E}|$.
5. find the large number inverse modulo the group order $q = |\mathcal{E}|$. We didn't implement the Montgomery batch inversion [4], but it can be easily implemented as the finite state machine is rather simple.

5.1 3-Montgomery-Core System

The 3-Montgomery-core system is implemented on ZedBoard™, on which a Z-7020 APSoC equivalent to an Artix®-7 FPGA is used. There are 53200 look-up tables and a Dual ARM® Cortex™-A9 MPCore™ processor on this APSoC [7], where the protocols (such as ECDH or ECDSA) is implemented on the ARM processor. A parameter setting the maximum compatible bit-size is configured in the 3-core system in our design. Here a 264-bit version and a 528-bit version are synthesized and tested with NIST curves, Brainpool curve P_{512} r1, and SEC P_{256} k1 curve [17] (a.k.a. the Bitcoin curve) are tested in both of the systems.

The resource requirement of the both systems is given as Table 3, and Table 4 shows the performances of the scalar multiplications on the systems. It should be noticed that there are two similar but different sorts of LUTs, so the LUT count of each module only implies the size scale of the module, and varies a little if the module is placed with different floor plans.

5.2 5-Montgomery-Core System

The 5-Montgomery-core system is implemented on ZC706 Evaluation Kit, on which a Z-7045 APSoC equivalent to a Kintex®-7 FPGA is used. There are 218600 LUTs and the same dual-core processor as in Z-7020 inside the APSoC

Table 3. Resource Requirement of our 3-Montgomery-core system on ZedBoard$^{\text{TM}}$. n in $\mathcal{S}_{3,n}$ implies the maximum n-bit compatibility design. No DSP slices are used.

Module	$\mathcal{S}_{3,264}$ $f_{\max} = 45$MHz			$\mathcal{S}_{3,528}$ $f_{\max} = 32.26$MHz		
	Slice LUT6	Slice Reg.	18Kb BRAM	Slice LUT6	Slice Reg.	18Kb BRAM
Mont. Mul. (each)	2057	280	0	3651	545	0
$xP^{-1}P$ Pool	1639	559	24	2764	1091	45
Setup Modules	1148	548	0	2085	1078	0
Diff. Adder	237	74	0	257	74	0
(X, Y, Z) Recovery	134	58	0	1853	58	0
Group Addition	248	54	0	178	54	0
Modular Inverse	523	303	0	1112	568	0
Element Check	208	27	0	139	27	0
Memory Pool	7895	1604	8	13510	3196	15
Misc. Modules	1682	1795	0	1556	3392	0
Total	19885	5862	32	35037	11172	60

Table 4. Performance of Scalar Multiplications $\mathbf{Q} = \langle k \rangle \mathbf{P}$ in various 3-core systems on ZedBoard$^{\text{TM}}$.

Elliptic Curve	$\mathcal{S}_{3,264}$ @ 45MHz		$\mathcal{S}_{3,528}$ @ 32.26MHz	
	Cycles	Time(ms)	Cycles	Time(ms)
NIST P_{224}	92402	2.053	138041	4.279
NIST P_{256}	105298	2.340	157273	4.875
SEC P_{256} k1 (BitCoin)	105298	2.340	157273	4.875
NIST P_{384}	-	-	233683	6.934
Brainpool P_{512} r1	-	-	311129	9.644
NIST P_{521}	-	-	316538	9.812

Table 5. Resource Requirement of our 5-Montgomery-core system on ZC706 Kit. n in $\mathcal{S}_{5,n}$ implies the maximum n-bit compatibility design. No DSP slices are used.

Module	$\mathcal{S}_{5,264}$ $f_{\max} = 83.33$MHz			$\mathcal{S}_{5,528}$ $f_{\max} = 62.50$MHz		
	Slice LUT6	Slice Reg.	18Kb BRAM	Slice LUT6	Slice Reg.	18Kb BRAM
Mont. Mul. (each)	2080	280	0	3455	545	0
$xP^{-1}P$ Pool	1635	559	40	3226	1096	75
Setup Modules	1079	548	0	1609	1078	0
Diff. Adder	247	103	0	1288	103	0
(X, Y, Z) Recovery	1395	76	0	1968	76	0
Group Addition	200	54	0	208	54	0
Modular Inverse	214	303	0	327	568	0
Element Check	137	27	0	162	27	0
Memory Pool	7662	2673	8	19554	5324	15
Misc. Modules	3973	1786	0	656	3412	0
Total	26941	7529	48	46269	14458	90

Table 6. Performance of Scalar Multiplications $\mathbf{Q} = \langle k \rangle \mathbf{P}$ in various 5-core systems on ZC706 Kit.

Elliptic Curve	$\mathcal{S}_{5,264}$ @ 83.33MHz		$\mathcal{S}_{5,528}$ @ 62.50MHz	
	Cycles	Time(ms)	Cycles	Time(ms)
NIST P_{224}	95657	1.148	133085	2.129
NIST P_{256}	109001	1.308	152429	2.439
SEC P_{256} k1 (BitCoin)	109001	1.308	152429	2.439
NIST P_{384}	-	-	226432	3.623
Brainpool P_{512} r1	-	-	301421	4.823
NIST P_{521}	-	-	306659	4.907

[7]. The resource requirements and time performances of the 264-bit version and the 528-bit version are given as Table 5 and Table 6.

Compared to 3-core systems, the cycles in our 5-core systems are not really better. It is because that the MUX/deMUX mechanism of the memory pool slows down all of the big integer operations, including the addition/subtraction. Each big integer operation takes 3 cycles to fetch and 6 to save the big integers (9 in total) in a 3-core system, but 5 and 10 (15 in total) in a 5-core system. When the interface is implemented with MUX/deMUX, it should be concerned that the cycles on the interfaces may make differences.

5.3 12-Montgomery-Core System

A 12-Montgomery-core system is implemented in our design to show the scalability of customized number of cores. However, we found that we can only implement a 12-core system with a maximum 264-bit size. 528-bit version can be synthesized, but will face a routing procedure failure due to routes too congested. The resource requirement and time performance of the 264-bit version is shown in Table 7 and 8.

MUX/deMUX problem on the memory pool will be more severe in the 12-core system, and a lot of Montgomery cores will be frequently useless during the computation. It is not practical to implement a 12-core system as one ECC engine.

5.4 3-Core vs 5-Core

Our ECC engine is designed as a custom IP to provide the hardware support of the ARM processor in Zynq-7000. A reasonable idea for the hardware/software co-design is to provide multiple ECC engines in the embedded system. We have run the implementation process to test how many ECC engines with our design can be put in the same system in ZC706 kit. The resource requirement of the multi-ECC-engine system is shown in Table 9.

We may apply the throughput-resource ratio blocks/(s × kLUT) to evaluate the effectiveness of the system we have built. The bigger the ratio is, the more

Table 7. Resource Requirement of our 12-Montgomery-core system $\mathcal{S}_{12,264}$ on ZC706 Kit. No DSP slices are used.

Module	$\mathcal{S}_{12,264}$, $f_{\max} = 45$MHz		
	Slice LUT6	Slice Reg.	18Kb BRAM
Mont. Mul. (each)	2052	280	0
$xP^{-1}P$ Pool	1339	559	96
Setup Modules	1081	548	0
Diff. Adder	5132	165	0
(X, Y, Z) Recovery	1109	74	0
Group Addition	183	54	0
Modular Inverse	211	303	0
Element Check	138	27	0
Memory Pool	18162	6366	8
Misc. Modules	2362	1987	0
Total	54337	1344	104

Table 8. Performance of Scalar Multiplications $\mathbf{Q} = \langle k \rangle \mathbf{P}$ in $\mathcal{S}_{12,264}$ with $f = 45$MHz on ZC706 Kit.

Elliptic Curve	Cycles	Time(ms)
NIST P_{224}	130513	2.900
NIST P_{256}	148721	3.305
SEC P_{256} k1 (BitCoin)	148721	3.305

effective the system is. We can see the TRR of $\mathcal{S}_{3,528}$, $\mathcal{S}_{5,528}$, $\mathcal{S}_{3,264}$, $\mathcal{S}_{5,264}$, and $\mathcal{S}_{12,264}$ are about 3.166, 2.472, 18.032, 11.825, and 4.800, respectively.

In a 5-core ECC engine there are sometimes some multipliers running dummy operations, so we can see the throughput-resource ratio is much lower. It is more effective to build 3-core engines in the system. Also we can see that a 12-core engine system is not effective.

6 Conclusion and Future Works

In this work, we have shown the power and the limit of multiple big-integer multiplication cores on the implementation of the Co-Z approach by Hutter *et al.* in elliptic curve cryptography. It is suggested to design a 3-Montgomery-core system to achieve the best performance, benchmarked as the throughput-resource ratio. We have also shown that it is possible to build a fast Montgomery ladder using the Co-Z approach with a 12-Montgomery-core system.

The system in our design can be improved in several ways. For the design of the block memory restoring the large numbers, the MUX/deMUX approach may be changed. The work by LaForest *et al.* [20–22] provides the solution saving the clock cycles reading and writing data from or into the memory, with the cost duplicated block memory modules being used. Also the design of the controller can be improved. The total finite state machine which constructs the controller

Table 9. Resource requirement and effectiveness of scalar multiplication for multi-ECC-engine systems on ZC706 Kit. f = 40MHz and NIST P_{521} curve applied for $\mathcal{S}_{n,528}$ and NIST P_{256} curve applied for $\mathcal{S}_{n,264}$.

ECC Engine	Count	Average LUT Count	System LUT Count	blocks/(s × kLUT)
$\mathcal{S}_{3,528}$ 7.913ms	2	37034	79634	3.174
	3	37290	119904	3.162
	4	38585	159788	3.163
	5	Fail (routes too congested)		
$\mathcal{S}_{5,528}$ 7.666ms	2	46264	100979	2.583
	3	51797	165830	2.360
	4	Fail (more than 218600)		
$\mathcal{S}_{3,264}$ 2.632ms	2	19602	42462	17.895
	3	19603	63389	17.981
	4	19601	84359	18.015
	5	19602	105323	18.037
	6	19601	126277	18.053
	7	19599	147247	18.062
	8	19598	168170	18.074
	9	19597	189087	18.084
	10	19596	210083	18.085
	11	Fail (more than 218600)		
$\mathcal{S}_{5,264}$ 2.725ms	2	26928	57150	12.842
	3	29499	95063	11.581
	4	26954	126859	11.571
	5	26604	158640	11.566
	6	26930	190417	11.563
	7	Fail (more than 218600)		
$\mathcal{S}_{12,264}$ 3.718ms	2	54332	112070	4.7999
	3	Fail (partial conflict)		

is huge. Since the controller controls the input and the output flows for all of the multipliers, it is possible to re-design the controller as several controllers, each of which controls only one multiplier.

References

1. Neal Koblitz: *Ellptic Curve Cryptosystems*, Mathematics of Computation **48:177**(1987), pp. 203-209.
2. Victor S. Miller: *Use of Elliptic Curves in Cryptography*, Crypto 1985, Lecture Notes in Computer Science v. 218, pp. 417-426, Springer.
3. Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters, *Twisted Edwards Curves*, Africacrypt 2008, Lecture Notes in Computer Science v. 5023, pp. 389–405, Springer.
4. Peter L. Montgomery: *Speeding the Pollard and elliptic curve methods of factorization*, Mathematics of Computation **48:177**(1987), pp. 243-264.
5. Peter L. Montgomery: *Modular Multiplication Without Trial Division*, Mathematics of Computation **44:170**(1985), pp. 519-521.
6. Ian Land, Ryan Kenny, Lance Brown, and Rob Pelt, *Shifting from Software to Hardware for Network Security*, White Paper, `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01261-shifting-from-software-to-hardware-for-network-security.pdf`, February 2016, Altera.
7. Zynq-7000 All Programmable SoCs Product Tables and Product Selection Guide, `http://www.xilinx.com/support/documentation/selection-guides/zynq-7000-product-selection-guide.pdf`, 2015, Xilinx.
8. Michael Hutter, Marc Joye, Yannick Sierra: *Memory-Constrained Implementations of Elliptic Curve Cryptography in Co-Z Coordinate Representation*, Africacrypt 2011, Lecture Notes in Computer Science v. 6737, pp. 170–187, Springer.
9. Henri Cohen, Atsuko Miyaji, Takatoshi Ono: *Efficient Elliptic Curve Exponentiation Using Mixed Coordinates*, Asiacrypt 1998, Lecture Notes in Computer Science v. 1514, pp. 51–65, Springer.
10. Paul C. Kocher: *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, Crypto 1996, Lecture Notes in Computer Science v. 1109, pp. 104-113, Springer.
11. Jean-Sébastien Coron: *Resistance Against Differential Power Analysis For Elliptic Curve Cryptosystems*, CHES 1999, Lecture Notes in Computer Science v. 1717, pp. 292-302, Springer.
12. Sung-Ming Yen, Marc Joye: *Checking before output may not be enough against fault-based cryptanalysis*, IEEE Trans. on Computers **49:9**(2000). pp. 967-970.
13. Margaux Dugardin, Louiza Papachristodoulou, Zakaria Najm, Lejla Batina, Jean-Luc Danger and Sylvain Guilley: *Dismantling real-world ECC with Horizontal and Vertical Template Attacks*, COSADE 2016. Lecture Notes in Computer Science v. 9689, pp. 88–108, Springer.
14. Daniel J. Bernstein, Tanja Lange: *Explicit-Formulas Database*, `https://hyperelliptic.org/EFD/`
15. National Institute of Standards and Technology: *Digital Signature Standard*, FIPS Publication 186-2, February 2000.
16. ECC Brainpool: *ECC Brainpool standard curves and curve generation.*, `http://www.ecc-brainpool.org/download/Domain-parameters.pdf`

17. Certicom Research: *SEC 2: Recommended Elliptic Curve Domain Parameters.* 2000.
18. Yu-Kwong Kwok, Ishfaq Ahmad: *Static scheduling algorithms for allocating directed task graphs to multiprocessors*, J. ACM CSUR **31:4**(1999), pp. 406–471.
19. Pedro Maat C. Massolino, Lejla Batina, Ricardo Chaves, Nele Mentens: *Low Power Montgomery Modular Multiplication on Reconfigurable Systems*, Cryptology ePrint Archive 2016/280.
20. Charles Eric LaForest, J. Gregory Steffan: *Efficient multi-ported memories for FPGAs*, Proc. ACM(SIGDA) FPGA 2010, pp. 41-50.
21. Charles Eric Laforest, Ming G. Liu, Emma Rae Rapati, J. Gregory Steffan: *Multi-ported memories for FPGAs via XOR*, Proc. ACM(SIGDA) FPGA 2012, pp. 209-218.
22. Charles Eric Laforest, Zimo Li, Tristan O'rourke, Ming G. Liu, J. Gregory Steffan: *Composing Multi-Ported Memories on FPGAs*, J. ACM Trans. Reconfig. Technol. Syst., **7:3**(2014), Article 16.
23. Samuel Neves, Mehdi Tibouchi: *Degenerate Curve Attacks*, PKC 2016, Lecture Notes in Computer Science v. 9615, pp. 19–35.

A Appendices

A.1 Critical Paths for Recovery of (X_1', Y_1', Z')

The task schedule graph of Algorithm 6 is shown in Figure 5(a), and the graph with ADD/SUB omitted is given in Figure 5(b).

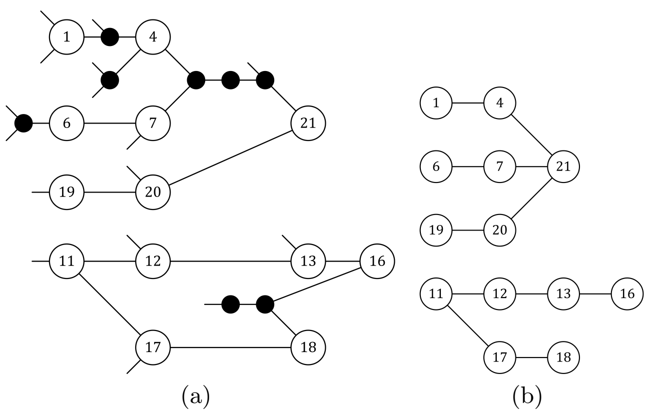


Fig. 5. The task schedule graph of Algorithm 6.

Observing in Figure 5(b), we can find the critical path of Algorithm 6 with respect to big-integer multiplication is $\langle 11 \to 12 \to 13 \to 16 \rangle$, and the length is 4. Recalling that $10\mathfrak{M} + 3\mathfrak{S}$ are required (13 in total), if Algorithm 6 is adopted to perform formula (5), it may be efficient to construct a 2-Montgomery-core, a 3-Montgomery-core, and a 4-Montgomery-core schedule.

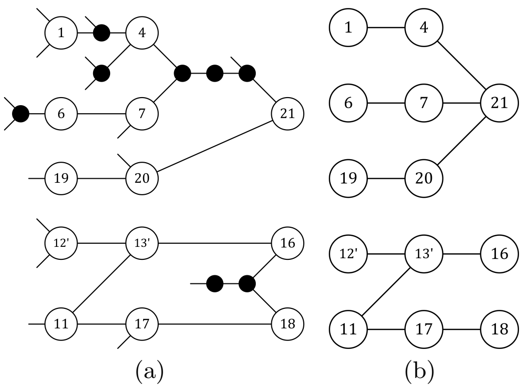


Fig. 6. The task schedule graph with step 12 and 13 modified in Algorithm 6.

$R_{11} \leftarrow T_P^2$	$R_{11} \leftarrow T_P^2$
$R_{12} \leftarrow X_1 \times R_{11}$	$R'_{12} \leftarrow X_1 \times x_P$
$R_{13} \leftarrow x_P \times R_{12}$	$R'_{13} \leftarrow R_{11} \times R'_{12}$
$R_{14} \leftarrow y_P + y_P$	$R_{14} \leftarrow y_P + y_P$
$R_{15} \leftarrow R_{14} + R_{14}$	$R_{15} \leftarrow R_{14} + R_{14}$
$X'_1 \leftarrow R_{15} \times R_{13}$	$X'_1 \leftarrow R_{15} \times R'_{13}$

It does not seem to be so efficient to construct a 5-Montgomery-core schedule, but actually the critical path can be further reduced. The related steps in Algorithm 6 are used to evaluate $X'_1 = 4y_P x_P T_P^2 X_1$. Observing that neither $x_P X_1$ nor $X_1 T_P^2$ will be used in other steps, the schedule of getting the result of $x_P T_P^2 X_1$ can be rearranged. We will get the product $x_P X_1$ first and then multiply T_P^2 with $x_P X_1$. The resulting task schedule graph is shown Figure 6(b), in which all critical paths have length 3. The 5-Montgomery-core schedule can be now created directly from Figure 6(b).

The 2-core and 3-core schedules for Algorithm 6 are shown as Algorithm 9 and 10, respectively. We can see that the 2-core schedule takes 7 rounds, while the 3-core, 5 rounds.

Now let's consider the task schedule graph, as shown in Figure 7(a) and 7(b), of Algorithm 5. The critical paths are of length 3 with respect to big-integer multiplications, so it is clear that the 2-Montgomery-core, the 3-Montgomery-core, and the 5-Montgomery-core schedules takes about 6, 5, and 3 rounds of big-integer multiplications, respectively. Because only step 1, 7, 15 (3 in total) can be performed in the first round, and there are still 9 multiplications to be performed *after* the first round, we can see that the 4-Montgomery-core schedule takes 4 rounds of big-integer multiplications.

An additional note is that when it is the case that a and $4b$ are constants, a 2-Montgomery-core system can perform Algorithm 5 with one round fewer than original case. This fact makes it more competitive than Algorithm 6.

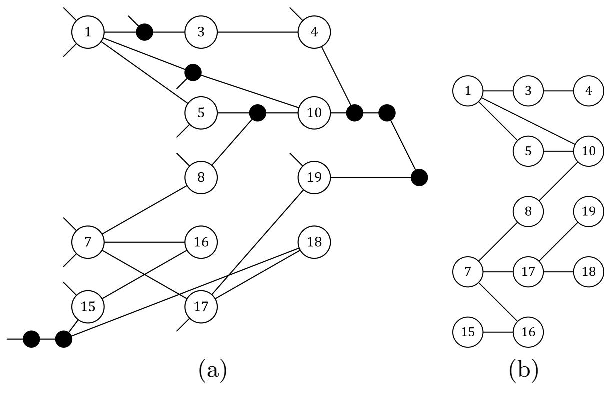


Fig. 7. The task schedule graph of Algorithm 5.

A.2 Design Detail for Montgomery Multipliers

Montgomery multiplication with reduction method [5] is given by Algorithm 2.

Algorithm 2: Montgomery multiplication with Reduction algorithm.

```
  Input: $A = a_{n-1}d^{n-1} + ...a_1d + a_0$, $B = b_{n-1}d^{n-1} + ...b_1d + b_0$,
         $P = p_{n-1}d^{n-1} + ...p_1d + p_0$, $d$, where $\gcd(P, d) = 1$
  Output: $C = c_{n-1}d^{n-1} + ...c_1d + c_0$
1 $p' \leftarrow -p_0^{-1} \mod d$
2 $C_0 \leftarrow 0$
3 for $i = 0$ to $n - 1$ do
4     $L_i \leftarrow C_i + a_iB$
5     $q_i \leftarrow L_i \mod d$
6     $C_{i+1} \leftarrow \frac{L_i + q_ip'P}{d}$
7 end
8 if $C_n \geq P$ then $C \leftarrow C_n - P$ else $C \leftarrow C_n$
9 return $C$
```

Two $(\lg d) \times (\lg P)$-bit and one $(\lg d) \times (\lg d)$-bit multiplication is used in this algorithm. There are a lot of works (e.g. [19] by Massolino *et al.* as the latest work as we know) to repeatedly utilize a $(\lg d) \times (\lg d)$-bit multiplier to construct a $(\lg d) \times (\lg P)$-bit multiplier, but in this work we build one $(\lg d) \times (\lg P)$ directly. It may be slow to build such a multiplier due to long critical paths and large delay, but the optimized multiplier size may not be fitted with the DSP slices only, but with the delay of other modules in the system. The best choice is to build a multiplier with similar delay compared to delays of other parts in the system. It is a good further research to do.

We do not use the DSP slices as they may be used for other purposes, especially multimedia. Another consideration not to use DSP slices is that we are going to build systems with a lot of Montgomery cores. With a $(\lg d) \times (\lg P)$-bit multiplier where $\lg P$ is large (at most 528 in our work), we may run out of

DSP slices very quickly when building a many-Montgomery-core system (especially the 12-core system in this work). Since DSP slices are not used, the base d applied in the algorithm cannot be large.

The $(\lg d) \times (\lg P)$-bit multiplier is utilized twice in Algorithm 2, but we can observe that in the case $q_i p' \times P$, the only varying parameter is q_i. Since d is small, it may be a good idea to build a table saving all of the possible $q_i p' P$ values. Such a table will cancel half of the utilization of the $(\lg d) \times (\lg P)$-bit multiplier, saving either half of the clock cycles or resources to build another $(\lg d) \times (\lg P)$-bit multiplier. In our work, the base d is selected as 2^8, and we have built a table with 256 entries to save all of the possible $q_i p' P$ values.

A.3 Algorithms

Algorithm 3: One-core addition-and-doubling algorithm in homogeneous projective co-Z coordinate - variant 1, for fixed ECC parameters. [8]

```
Input: $X_1$, $X_2$, $Z$, $x_P$, $a$, $4b$
Output: $X'_1$, $X'_2$, $Z'$
1  $R_1 \leftarrow Z^2$
2  $R_2 \leftarrow a \times R_1$
3  $R_3 \leftarrow Z \times R_1$
4  $R_4 \leftarrow 4b \times R_3$
5  $R_5 \leftarrow X_2^2$
6  $R_6 \leftarrow R_5 - R_2$
7  $R_7 \leftarrow R_6^2$
8  $R_8 \leftarrow R_5 + R_2$
9  $R_9 \leftarrow X_2 \times R_8$
10 $R_{10} \leftarrow R_9 + R_9$
11 $R_{11} \leftarrow R_{10} + R_{10}$
12 $R_{12} \leftarrow R_{11} + R_4$
13 $R_{13} \leftarrow R_8 + R_2$
14 $R_{14} \leftarrow X_1^2$
15 $R_{15} \leftarrow R_{13} + R_{14}$
16 $R_{16} \leftarrow X_1 - X_2$
17 $R_{17} \leftarrow X_2 + X_2$
18 $R_{18} \leftarrow R_{17} \times R_4$
19 $R_{19} \leftarrow R_7 - R_{18}$
20 $R_{20} \leftarrow R_{16}^2$
21 $R_{21} \leftarrow R_{15} - R_{20}$
22 $R_{22} \leftarrow R_{16} + R_{17}$
23 $R_{23} \leftarrow R_{22} \times R_{21}$
24 $R_{24} \leftarrow R_{23} + R_4$
25 $R_{25} \leftarrow Z \times R_{20}$
26 $R_{26} \leftarrow x_P \times R_{25}$
27 $R_{27} \leftarrow R_{24} - R_{26}$
28 $X'_1 \leftarrow R_{27} \times R_{12}$
29 $X'_2 \leftarrow R_{20} \times R_{19}$
30 $Z' \leftarrow R_{25} \times R_{12}$
31 return $(X'_1, X'_2, Z')$
```

Algorithm 4: One-core addition-and-doubling algorithm in homogeneous projective co-Z coordinate - variant 2, optimized version for dynamic ECC parameters. [8]

```
Input: $X_1$, $X_2$, $T_P = x_P Z$, $T_a = aZ^2$, $T_b = 4bZ^3$
Output: $X'_1$, $X'_2$, $T'_P$, $T'_a$, $T'_b$
1  $R_1 \leftarrow X_1 - X_2$
2  $R_2 \leftarrow R_1^2$
3  $R_3 \leftarrow X_2^2$
4  $R_4 \leftarrow R_3 - T_a$
5  $R_5 \leftarrow R_4^2$
6  $R_6 \leftarrow X_2 + X_2$
7  $R_7 \leftarrow R_6 \times T_b$
8  $R_8 \leftarrow R_5 - R_7$
9  $R_9 \leftarrow R_6 + R_6$
10 $R_{10} \leftarrow R_3 + T_a$
11 $R_{11} \leftarrow R_9 \times R_{10}$
12 $R_{12} \leftarrow R_{11} + T_b$
13 $R_{13} \leftarrow X_1 + X_2$
14 $R_{14} \leftarrow R_{10} + T_a$
15 $R_{15} \leftarrow R_{14} - R_2$
16 $R_{16} \leftarrow X_1^2$
17 $R_{17} \leftarrow R_{15} + R_{16}$
18 $R_{18} \leftarrow R_{13} \times R_{17}$
19 $R_{19} \leftarrow R_{18} + T_b$
20 $R_{20} \leftarrow R_{12} \times R_{19}$
21 $X'_2 \leftarrow R_2 \times R_8$
22 $R_{22} \leftarrow R_2 \times R_{12}$
23 $R_{23} \leftarrow R_{22} \times T_b$
24 $R_{24} \leftarrow R_{22}^2$
25 $T'_P \leftarrow T_P \times R_{22}$
26 $T'_a \leftarrow T_a \times R_{24}$
27 $T'_b \leftarrow R_{23} \times R_{24}$
28 $X'_1 \leftarrow R_{20} - T'_P$
29 return $(X'_1, X'_2, T'_P, T'_a, T'_b)$
```

Algorithm 5: One-core (X, Y, Z)-recovery algorithm in homogeneous projective co-Z coordinate - variant 1, for fixed ECC parameters. [8]

```
**Input:** $X_1$, $X_2$, $Z$, $x_P$, $y_P$, $a$, $4b$
**Output:** $X_1'$, $Y_1'$, $Z'$
1  $R_1 \leftarrow x_P \times Z$
2  $R_2 \leftarrow X_1 - R_1$
3  $R_3 \leftarrow R_2^2$
4  $R_4 \leftarrow R_3 \times X_2$
5  $R_5 \leftarrow R_1 \times X_1$
6  $R_6 \leftarrow X_1 + R_1$
7  $R_7 \leftarrow Z^2$
8  $R_8 \leftarrow a \times R_7$
9  $R_9 \leftarrow R_5 + R_8$
10 $R_{10} \leftarrow R_6 \times R_9$
11 $R_{11} \leftarrow R_{10} - R_4$
12 $R_{12} \leftarrow R_{11} + R_{11}$
13 $R_{13} \leftarrow y_P + y_P$
14 $R_{14} \leftarrow R_{13} + R_{13}$
15 $R_{15} \leftarrow R_{14} \times X_1$
16 $X_1' \leftarrow R_{15} \times R_7$
17 $R_{17} \leftarrow R_7 \times Z$
18 $Z' \leftarrow R_{17} \times R_{14}$
19 $R_{19} \leftarrow 4b \times R_{17}$
20 $Y_1' \leftarrow R_{19} + R_{12}$
21 **return** $(X_1', Y_1', Z')$
```

Algorithm 6: One-core (X, Y, Z)-recovery algorithm in homogeneous projective co-Z coordinate - variant 2, optimized for dynamic ECC parameters. [8]

```
**Input:** $X_1$, $X_2$, $T_P = x_P Z$, $T_a = aZ^2$, $T_b = 4bZ^3$, $x_P$, $y_P$
**Output:** $X_1'$, $Y_1'$, $Z'$
1  $R_1 \leftarrow T_P \times X_1$
2  $R_2 \leftarrow R_1 + T_a$
3  $R_3 \leftarrow X_1 + T_P$
4  $R_4 \leftarrow R_2 \times R_3$
5  $R_5 \leftarrow X_1 - T_P$
6  $R_6 \leftarrow R_5^2$
7  $R_7 \leftarrow R_6 \times X_2$
8  $R_8 \leftarrow R_4 - R_7$
9  $R_9 \leftarrow R_8 + R_8$
10 $R_{10} \leftarrow R_9 + T_b$
11 $R_{11} \leftarrow T_P^2$
12 $R_{12} \leftarrow X_1 \times R_{11}$
13 $R_{13} \leftarrow x_P \times R_{12}$
14 $R_{14} \leftarrow y_P + y_P$
15 $R_{15} \leftarrow R_{14} + R_{14}$
16 $X_1' \leftarrow R_{15} \times R_{13}$
17 $R_{17} \leftarrow R_{11} \times T_P$
18 $Z' \leftarrow R_{15} \times R_{17}$
19 $R_{19} \leftarrow x_P^2$
20 $R_{20} \leftarrow R_{19} \times x_P$
21 $Y_1' \leftarrow R_{20} \times R_{10}$
22 **return** $(X_1', Y_1', Z')$
```

Algorithm 7: Two-core addition-and-doubling algorithm in homogeneous projective co-Z coordinate, optimized for dynamic ECC parameters.

```
**Input:** $X_1$, $X_2$, $T_P = x_P Z$, $T_a = aZ^2$, $T_b = 4bZ^3$
**Output:** $X_1'$, $X_2'$, $T_P'$, $T_a'$, $T_b'$
1  $R_3 \leftarrow X_2^2$; $R_2 \leftarrow X_1^2$
2  $R_1 \leftarrow X_1 - X_2$; $R_4 \leftarrow R_3 - T_a$
3  $R_1 \leftarrow R_1^2$; $R_4 \leftarrow R_4^2$
4  $R_5 \leftarrow X_2 + X_2$; $R_3 \leftarrow R_3 + T_a$
5  $X_2' \leftarrow R_5 + R_5$; $X_1' \leftarrow X_1 + X_2$
6  $X_2' \leftarrow R_5 \times T_b$; $R_5 \leftarrow X_2' \times R_3$
7  $X_2' \leftarrow R_4 - X_2'$; $R_4 \leftarrow R_5 + T_b$
8  $X_2' \leftarrow R_1 \times X_2'$; $R_5 \leftarrow R_1 \times R_4$
9  $R_3 \leftarrow R_3 + T_a$; $R_2 \leftarrow R_2 - R_1$
10 $R_3 \leftarrow R_3 + R_2$
11 $R_2 \leftarrow R_5 \times T_b$; $R_1 \leftarrow R_5^2$
12 $T_P' \leftarrow T_P \times R_5$
13 $X_1' \leftarrow X_1' \times R_3$; $T_a' \leftarrow T_a \times R_1$
14 $X_1' \leftarrow X_1' + T_b$
15 $X_1' \leftarrow R_4 \times X_1'$; $T_b' \leftarrow R_2 \times R_1$
16 $X_1' \leftarrow X_1' - T_P'$
17 **return** $(X_1', X_2', T_P', T_a', T_b')$
```

Algorithm 8: Three-core addition-and-doubling algorithm in homogeneous projective co-Z coordinate, optimized for dynamic ECC parameters.

```
   Input: $X_1$, $X_2$, $T_P = x_P Z$, $T_a = aZ^2$, $T_b = 4bZ^3$
   Output: $X'_1$, $X'_2$, $T'_P$, $T'_a$, $T'_b$
1  $R_0 \leftarrow X_1 - X_2$; $R_1 \leftarrow X_2 + X_2$
2  $R_2 \leftarrow R_0^2$; $R_0 \leftarrow X_2^2$; $R_3 \leftarrow T_b \times R_1$
3  $R_0 \leftarrow R_0 - T_a$; $R_1 \leftarrow R_1 + R_1$; $R_4 \leftarrow R_0 + T_a$
4  $R_0 \leftarrow R_0^2$; $R_5 \leftarrow R_1 \times R_4$; $R_1 \leftarrow X_1^2$
5  $R_3 \leftarrow R_0 - R_3$; $R_1 \leftarrow R_4 + T_a$; $R_4 \leftarrow R_1 - R_2$
6  $R_0 \leftarrow R_5 + T_b$; $R_1 \leftarrow X_1 + X_2$; $R_4 \leftarrow R_1 + R_4$
7  $R_2 \leftarrow R_1 \times R_4$; $X'_2 \leftarrow R_2 \times R_3$; $R_3 \leftarrow R_2 \times R_0$
8  $R_2 \leftarrow R_2 + T_b$
9  $R_2 \leftarrow R_0 \times R_2$; $R_1 \leftarrow R_3 \times T_b$; $R_4 \leftarrow R_3^2$
10 $T'_P \leftarrow T_P \times R_3$; $T'_a \leftarrow T_a \times R_4$; $T'_b \leftarrow R_1 \times R_4$
11 $X'_1 \leftarrow R_2 - T'_P$
12 return $(X'_1, X'_2, T'_P, T'_a, T'_b)$
```

Algorithm 9: Two-core (X, Y, Z)-recovery algorithm in homogeneous projective co-Z coordinate, optimized for dynamic ECC parameters.

```
   Input: $X_1$, $X_2$, $T_P = x_P Z$, $T_a = aZ^2$, $T_b = 4bZ^3$, $x_P$, $y_P$
   Output: $X'_1$, $Y'_1$, $Z'$
1  $R_2 \leftarrow X_1 - T_P$
2  $R_1 \leftarrow T_P \times X_1$; $R_2 \leftarrow R_2^2$
3  $R_1 \leftarrow R_1 + T_a$; $R_3 \leftarrow X_1 + T_P$
4  $R_1 \leftarrow R_1 \times R_3$; $R_2 \leftarrow R_2 \times X_2$
5  $R_1 \leftarrow R_1 - R_2$
6  $R_2 \leftarrow T_P^2$; $Y'_1 \leftarrow x_P^2$
7  $X'_1 \leftarrow X_1 \times R_2$; $Z' \leftarrow R_2 \times T_P$
8  $X'_1 \leftarrow x_P \times X'_1$; $Y'_1 \leftarrow Y'_1 \times x_P$
9  $R_1 \leftarrow R_1 + R_1$; $R_2 \leftarrow y_P + y_P$
10 $R_1 \leftarrow R_1 + T_b$; $R_2 \leftarrow R_2 + R_2$
11 $X'_1 \leftarrow X'_1 \times R_2$; $Z' \leftarrow R_2 \times Z'$
12 $Y'_1 \leftarrow R_1 \times Y'_1$
13 return $(X'_1, Y'_1, Z')$
```

Algorithm 10: Three-core (X, Y, Z)-recovery algorithm in homogeneous projective co-Z coordinate, optimized for dynamic ECC parameters.

```
   Input: $X_1$, $X_2$, $T_P = x_P Z$, $T_a = aZ^2$, $T_b = 4bZ^3$, $x_P$, $y_P$
   Output: $X'_1$, $Y'_1$, $Z'$
1  $R_0 \leftarrow X_1 - T_P$
2  $R_1 \leftarrow R_0^2$; $R_2 \leftarrow T_P \times X_1$
3  $R_0 \leftarrow X_1 + T_P$; $R_2 \leftarrow R_2 + T_a$
4  $R_0 \leftarrow R_1 \times X_2$; $R_2 \leftarrow R_2 \times R_0$; $R_1 \leftarrow T_P \times T_P$
5  $R_2 \leftarrow R_2 - R_0$
6  $R_0 \leftarrow x_P \times x_P$; $R_4 \leftarrow X_1 \times R_1$; $R_3 \leftarrow R_1 \times T_P$
7  $R_1 \leftarrow R_0 \times x_P$; $R_4 \leftarrow R_4 \times x_P$
8  $R_0 \leftarrow R_2 + R_2$; $R_2 \leftarrow y_P + y_P$
9  $R_0 \leftarrow R_0 + T_b$; $R_2 \leftarrow R_2 + R_2$
10 $Y'_1 \leftarrow R_1 \times R_0$; $X'_1 \leftarrow R_2 \times R_4$; $Z' \leftarrow R_2 \times R_3$
11 return $(X'_1, Y'_1, Z')$
```