Efficient FPGA Design of Exception-Free Generic Elliptic Curve Cryptosystems

Kiyofumi Tanaka[1(✉)], Atsuko Miyaji[1,2], and Yaoan Jin[2]

[1] School of Information Science, Japan Advanced Institute of Science and Technology, 1–1 Asahidai, Nomi, Ishikawa 923–1292, Japan
kiyofumi@jaist.ac.jp

[2] Graduate School of Engineering, Osaka University, Suita, Osaka 565–0871, Japan
miyaji@comm.eng.osaka-u.ac.jp, jin@cy2sec.comm.eng.osaka-u.ac.jp

Abstract. Elliptic curve cryptography (ECC) is one of promising cryptosystems in embedded systems as it provides high security levels with short keys. Scalar multiplication is a dominating and time-consuming process that ensures security in ECC. We implement hardware modules for generic ECC over 256-bit prime fields on field-programmable gate array (FPGA). The key points in our design are (1) secure and exception-free for any scalar with less memory usage, (2) long-bit modular arithmetic modules utilizing today's advanced and high-performance programmable logic and considering balance between the modules in terms of propagation delay, (3) parallelism extraction inside each elliptic curve point computation as well as between the point computations, and (4) efficient hardware–software co-processing facilitated by application interfaces between a processing core and hardware modules. The evaluation results demonstrate that our design achieves the best performance to existing FPGA designs without using a table for generic ECC.

Keywords: Elliptic curve cryptosystem · Complete addition · Exception-free · FPGA

1 Introduction

Elliptic curve cryptography (ECC) is one of promising cryptosystems in embedded systems as it provides high security levels with short keys. Therefore, ECC is becoming a mainstream cryptosystem in embedded systems where memory resources are constrained. However, the use of ECC still requires considerable processing time as well as memory, especially for software in embedded systems with constrained processing speed. Hardware acceleration is a promising option to reduce the overhead of software processing.

The dominant computation of ECCs is scalar multiplication, which computes kP for an elliptic curve point P and a scalar k. Thus, the security and efficiency of the scalar multiplication are paramount. To implement scalar multiplication, several types of coordinates for elliptic curves exist (such as affine, Jacobian, or

© Springer Nature Switzerland AG 2021
K. Sako and N. O. Tippenhauer (Eds.): ACNS 2021, LNCS 12726, pp. 393–414, 2021.
https://doi.org/10.1007/978-3-030-78372-3_15

Projective). To be secure against simple power analysis (SPA), these coordinates need to be combined with secure scalar multiplication algorithms without any branch instruction such as Joye's RL algorithm [17].

Recently, more advanced security notion of *exception free* is introduced [32], where scalar multiplication should work for any scalar k including $k = 0$. Since complete addition (CA) formulae can work in the same formulae of addition and doubling formulae [32], combining with Joye's RL algorithm is secure against SPA and exception-free for any scalar k. Although various ECC FPGA implementations have been proposed so far [6–8,12–14,22,26], any of them neither employs CA formulae nor satisfies exception-free secure. They fail to execute a case that the MSB of k is equal to 0 as well as $k = 0$. However, CA formulae uses three coordinates of X, Y, and Z to represent an elliptic curve point and, thus, it is far from less memory. Recently, another approach to use exception-free affine coordinate, which is a combination of affine and extended affine coordinates, is proposed [16]. Combining exception-free affine coordinates with improved Joye's RL algorithm is secure against SPA for any scalar k. Importantly, exception-free affine coordinates can represent an elliptic curve point by two coordinates, which can work with less memory compared with combination of CA formulae and Joye's RL algorithm. To give high performance of the scalar multiplication while keeping the resistance to SPA, one of simple ways is to focus on a specified elliptic curve such as NIST P-256 [14], which chooses an affine coordinate for the reason of less memory and works efficiently on only NIST P-256. However, their design cannot be applied to any other elliptic curves. For universal usage, an important point is *generic* elliptic curve design, which provides an architecture available to any elliptic curve over a finite field. Another strategy to give the high-performance is to use a precomputation table such as window methods [25,30]. However, it requires additional memory. For example, the implementation in [24] can work on a generic elliptic curve and is secure against SPA and exception-free. However, since it is based on window methods, it needs additional memory of points.

In this paper, we aim at efficient hardware–software FPGA design of generic ECC with less-memory which is secure against SPA and satisfies exception-free for $\forall k$. Especially, we focus on system-on-chip (SoC) type of FPGA device. Compared to conventional FPGA devices with programmable logic only, SoC FPGA device provides a tightly coupled system so that data transfer between a processor core and a programmable logic part is performed at high speeds. We implement EC point computations as hardware modules by making complete use of advanced and high-performance programmable logic in today's FPGA devices. Software processing performs scalar multiplication by invoking each hardware module when needed. In real-time applications, EC scalar multiplication processing shorter than 1 ms is highly desirable as many control tasks use a period less than or equal to 1 ms [34]. Our design achieves this requirement by utilizing high-performance resources in today's FPGA devices. To the best of our knowledge, our accelerator performs secure and exception-free scalar multiplications

faster than any FPGA implementations for a generic ECC without any table over 256-bit prime fields [6,7,12–14,22,26].

This paper is organized as follows. First, we summarize basic notion of elliptic cryptosystems in Sect. 2. Then, we describe related works in Sect. 3. Subsequently, we clarify our targets in Sect. 4. We describe the details of our design in Sect. 5. Experimental results are shown in Sect. 6. We conclude our work in Sect. 7.

2 Operations in Elliptic Curve Cryptography

2.1 Addition Formulae on Elliptic Curve

Elliptic curve cryptography (ECC), which was proposed in the 1980s [19,29], is a public-key cryptography system, and its cipher strength depends on the difficulty of the elliptic curve discrete logarithm problem. This section describes our target elliptic curve and addition formulae.

We target the elliptic curve E over a prime field $\mathbb{F}_p (p > 3)$ expressed by the following short Weierstrass form:

$$E/\mathbb{F}_p : y^2 = x^3 + ax + b \quad (a, b \in \mathbb{F}_p, 4a^3 + 27b^2 \neq 0)$$

For a set of points on this curve and a point at infinity, $\mathcal{O}$, the addition is geometrically defined. In the affine coordinate system, for a point $P_1 = (x_1, y_1)$ and a point $P_2 = (x_2, y_2)$ $(P_1 \neq P_2)$, point addition, $P_3 = (x_3, y_3) = P_1 + P_2$, is calculated as follows:

$$x_3 = (\frac{y_2 - y_1}{x_2 - x_1})^2 - x_1 - x_2$$
$$y_3 = (\frac{y_2 - y_1}{x_2 - x_1})(x_1 - x_3) - y_1$$

Similarly, point doubling, $P_3 = 2P_1$, is defined as follows:

$$x_3 = (\frac{3x_1^2 + a}{2y_1})^2 - 2x_p$$
$$y_3 = (\frac{3x_1^2 + a}{2y_1})(x_1 - x_3) - y_1$$

Compared to the formulae for other projective coordinate systems, the above calculation of the affine coordinate system is desirable in terms of memory usage. As the variables in the above formulae (x_1, y_2, and so on) are multi-bit data (longer than 32- or 64-bit) and P_3 must be an element in $\mathbb{F}_p$, multi-bit modular addition (subtraction), multi-bit modular multiplication, and multi-bit modular inversion for division are required. Performing multi-bit division directly would involve high computational complexity. Instead, an inverse element should be obtained and multiplied.

2.2 Scalar Multiplication

While processing encryption and decryption in elliptic curve cryptography systems, multiplication of a point on the elliptic curve and scalar dominates the total computation cost. Scalar multiplication can be performed by applying point addition and point doubling in Sect. 2.1. The simplest approach is to use a binary method [10]: variables R and Q, initialized to $\mathcal{O}$ and P, respectively, are prepared. A scalar value in binary is scanned from the least significant bit to the most significant bit (or in the opposite direction). When the corresponding bit is zero, Q is updated by $2Q$. Otherwise, Q is updated by $2Q$ after R is updated by $R + Q$. (For the opposite scanning direction, R is updated by addition after Q is updated by doubling.) The final R is the result of scalar multiplication, kP.

In the above method, the execution time of a loop iteration varies according to the value of the corresponding bit. That is, it performs only point doubling, or both point addition and point doubling. This indicates that the method is vulnerable to simple power analysis (SPA) attacks that exploit energy dissipation measured and infer the input value [20]. To alleviate this problem, Joye's m-ary Ladder [17] was proposed by reforming the binary method, where the computation in an iteration is made uniform. This uniformity is achieved by transforming each digit in the m-ary expression of the scalar into a nonzero form such that an addition and a m-times multiplication are executed every time. However, it is not completely secure against SPA, as there can be exceptional addition with $\mathcal{O}$ in the processing of point addition on the affine coordinates.

To solve the problem of exceptional addition with $\mathcal{O}$, Jin et al. proposed the New 1-bit 2-ary Right-to-Left Powering Ladder [16]. This algorithm does not involve exceptional addition by avoiding initialization with $\mathcal{O}$ in the above-mentioned method. In addition, the algorithm is extended to the New 2-bit 2-ary Right-to-Left Powering Ladder, where the main loop is unrolled such that processing for every two bits in k is done in an iteration. In the iteration, affine double-quadruple [23] is used to obtain double and quadruple values, with only one inversion computation involved. As a result, the combination of loop unrolling and affine double-quadruple reduces the amount of computation in the main loop.

To reduce the amount of hardware resources required, our target algorithm is the above-mentioned New 1-bit 2-ary Right-to-Left Powering Ladder, rather than the 2-bit derivation. We implement a scalar multiplication accelerator by analyzing dataflow in point addition and point doubling and extracting full parallelism in the algorithm, as well as fully utilizing advanced and high-performance FPGA logic resources.

3 Related Work

In this section, we review several FPGA implementations of scalar multiplication. FPGA implementations over prime fields are classified into specific-prime-field and general-prime-field. The examples of the former are observed in

[2,9,26,27,33]. The use of specific primes enables fast modular reduction, leading to division replaced by a series of additions and subtractions; however, it has some inflexibility, where the accelerators are limited to supporting specific prime fields, for example, the NIST primes, i.e., generalized Merssene primes. In contrast, implementations over general prime fields provide considerable flexibility in terms of selecting a prime number, p, and they can be applied to various applications such as digital signature generation and key agreement. Some examples of FPGA implementations over general-prime-fields include [6,7,12–14,22].

Another perspective is regarding the choice of coordinate system. One is an affine coordinate system, and the other is a projective or Jacobian coordinate system. The former has the advantage of smaller memory usage than the latter with an additional axis of coordinates, leading to a smaller number of registers in FPGA resources. However, inversion calculation is required for every point addition and doubling over affine coordinates, whereas it is performed once at the end of scalar multiplication over projective or Jacobian coordinates. In this study, because we prioritize the flexibility of prime fields and memory/hardware resource usage, we focus on FPGA designs for scalar multiplication over general prime fields and affine coordinates.

Ghosh et al. proposed an FPGA implementation that performs point addition and point doubling in parallel [6]. This parallelism is naive and common in various hardware implementations. For modular multiplication, their interleaved multiplication algorithm takes $k+1$ cycles, where k is the bit length of p, i.e., 257 cycles when 256-bit p is assumed. Similarly, the modular multipliers in [12,13,22] used similar algorithms and took at least k cycles. As modular multiplication is one of the main calculations in EC scalar multiplication, the cycles taken by these implementations result in long computation time for scalar multiplication. In contrast, our implementation of modular multiplication described in Sect. 5, which utilizes DSP modules embedded in FPGA devices, takes 28 cycles with 256-bit p, leading to a much faster execution of EC scalar multiplication. Today's FPGA devices include high-performance embedded DSP modules. Using them is better than constructing complicated multipliers with programmable logic based on look-up tables and long-delayed wiring.

Javeed et al. used a modular multiplier that includes a radix-8 Booth encoded multiplier with iterative addition and reduction modulo p of partial products [14]. Although this modular multiplier reduces the execution cycles compared with the above-mentioned modular multipliers, it still requires 88 cycles for 256-bit p, whereas our implementation requires only 28 cycles. In addition, the cascaded adders structure in [14] makes the critical path long, thereby preventing the clock frequency from improving.

In the implementations in [26,33] (which are dedicated to NIST P-256 prime fields), redundant signed digit (RSD) arithmetic is used, where multi-bit addition can be performed without carry propagation at the expense of additional FPGA resource areas. In contrast, our implementation of addition/subtraction simply utilizes fast carry logic in today's FPGA slices and achieves one-cycle addition/subtraction with 256-bit operands at over 200 MHz. Similar to the use

of DSP modules, we can expect that using fast carry logic yields faster and smaller adders than RSD-based adders.

Considering the use in embedded systems, the amount of hardware required for EC accelerators is important. The accelerator in [7] saves on hardware by sharing hardware resources among different finite-field modular arithmetic operations and among EC point computations at the sacrifice of parallelism inside point addition and doubling. In contrast, our implementation decouples multi-bit arithmetic units from EC point computations to reuse arithmetic units between point computations that are serially processed while duplicating the arithmetic units to extract full parallelism inside each point computation and among point computations running in parallel.

Unlike the above-mentioned designs, some implementations use projective or Jacobian coordinate systems. The use of these coordinate systems eliminates division in every EC point addition/doubling at the cost of increased storage space, accelerating scalar multiplication. The implementations in [8,24] are examples. These designs can be applicable to real-time applications since they achieve less than 1 ms of processing for scalar multiplication, whereas all the designs in affine coordinates mentioned above take more than 2 ms. Our objective is to achieve less than 1 ms of processing for scalar multiplication in the affine coordinate system to satisfy the performance and cost requirements in various real-time applications.

4 Target Algorithm and Modular Arithmetics

4.1 Algorithms for Scalar Multiplication

Algorithm 1 proposed in [16] computes a scalar multiplication with a point, P, on the elliptic curve and an integer scalar, k, and outputs $Q = kP$. This algorithm has two important features. First, an affine coordinate is used to reduce the memory usage. Second, it satisfies *secure generality* (i.e., it can operate on any input scalar k). To achieve secure generality, this algorithm does not include exceptional initialization or exceptional computation and is thus secure against a side-channel attack (SCA). Therefore, we choose this algorithm.

The algorithm comprises three parts: initialization, main loop, and final correction. The initialization starts with $R[0] =\leftarrow -P$ and $R[1] \leftarrow P$ in Steps 1 and 2, respectively, avoiding exceptional initialization with $\mathcal{O}$ and exceptional computation $\mathcal{O} + P$ in the main loop while leaving the computation for adjustment, $+2R[1]$, in Step 10 in the final correction. Steps 3 and 4 perform affine point doubling and affine point addition, respectively, and help in avoiding exceptional computations, $P + P$ or $P - P$. The extra computations are adjusted in the final correction. The main loop from Steps 5 to 8 dominates the execution time of scalar multiplication. In each iteration, affine point addition and affine point doubling are performed in Steps 6 and 7, respectively.

After the main loop, the final correction is performed. This is one of the important security-enhanced parts. They introduced extended affine point addition and doubling that can execute exceptional computation such as $P - P$ and

Algorithm 1. New 1-bit 2-ary Rright-to-Left Powering Ladder (Algorithm 7 in [16])

Input: $P \in E(\mathbb{F}_q)$, $k = \sum_{i=0}^{l-1} k_i 2^i$, $k \in [0, N]$
Output: $Q = kP$

Initialization
1: $R[0] \leftarrow -P$
2: $R[1] \leftarrow P$
3: $A \leftarrow 2P$
4: $R[k_0] \leftarrow R[k_0] + A$

Main Loop
5: **for** $i = 1$ **to** $l - 1$ **do**
6: $R[k_i] \leftarrow R[k_i] + A$
7: $A \leftarrow 2A$
8: **end for**

Final Correction
9: $R[k_0] \leftarrow R[k_0] - P$
10: $A \leftarrow (-A + R[0]) +_E 2_E R[1]$
11: **return** A

$2P = \mathcal{O}$. In Step 9, an affine point addition is applied to $R[k_0]$ and the complement of P. The conventional affine coordinates are used in Steps 1 to 9. In contrast, in Step 10, while affine point addition, $-A + R[0]$, is computed using conventional affine coordinates, extended affine point doubling is applied to $R[1]$, described by $2_E R[1]$. Finally, these two results are added by an extended affine point addition ($+_E$ in Step 10). Extended affine point addition and extended affine point doubling are used to avoid exceptional computations. Details regarding the extended affine point addition and affine point doubling can be found in [16]. Importantly, our elegant FPGA design does not increase the FPGA resource usage as arithmetic calculators in the extended affine point addition and doubling are shared with other point computations.

4.2 Modular Arithmetic

Elliptic curve (EC) point computations comprise multi-bit modular arithmetic: addition, subtraction, multiplication, and inversion. Modular addition (subtraction) is a combination of multi-bit addition (subtraction) and conditional subtraction (addition) for the residue. In our implementation, subtraction (addition) for the residue is performed only when it is indispensable. Each addition or subtraction is performed without the residue, thereby reducing the computation complexity at the expense of additional most-significant bits. After several additions/subtractions, subtraction or addition for the residue is applied to reduce the value to the field range.

For modular multiplication, we use the Montgomery multiplication algorithm [31] that involves multi-bit additions/subtractions and multi-bit multiplications.

Several algorithms, such as the Karatsuba method [18], are candidates for multi-bit multiplications. We adopt a simple method for the parallel generation of partial products as the operand length is at most 264 bits. For inversion calculation, a constant binary extended GCD algorithm [3] is selected, wherein the shift and subtraction operations are performed in each iteration, while the number of iterations is constant.

5 Design and Implementation

Our method offers programmability (i.e., application programming interfaces) in designing an accelerator for scalar multiplications such that higher-layer EC cryptography protocols such as ECDH and ECDSA [4] can invoke the hardware accelerator when required. EC point computations along with modular/multi-bit arithmetic calculators are provided as hardware modules, whereas the control sequence among the modules is provided via software processing, which achieves hardware–software co-processing for scalar multiplication. Current tightly coupled SoC-type FPGA devices facilitate fast communication between processor cores and FPGA modules, thereby making hardware–software co-processing efficient.

5.1 Design of Arithmetic Units

Based on the directions mentioned in the previous section, we designed and implemented the following arithmetic units in hardware description language (VHDL): multi-bit adder, multi-bit subtractor, multi-bit multiplier, modular Montgomery multiplier, and modular inversion calculator.

Multi-bit Adder/Subtracter. While our ECC system targets 256-bit elements over the prime field $\mathbb{F}_p$ with 256-bit p, 260-bit adders and subtractors are implemented since 260-bit temporary data emerge internally as the result of postponed residue operations. In addition, a 520-bit adder is required inside Montgomery multipliers, wherein multi-bit multiplication generates 520-bit operands. Furthermore, a 264-bit subtractor is used in Montgomery multipliers. These adders and subtracters are designed to output the results in one clock cycle, as current FPGA devices include fast carry logic and can achieve one-cycle 520-bit addition at over 200 MHz. Simple '+' operators in the HDL source files generate these calculators.

Multi-bit Multiplier. In the process of Montgomery multiplications, 264-bit $\times$ 264-bit multiplications are performed. Our multi-bit multiplier generates a multiplication result in four clock cycles, as depicted in Fig. 1. In the first cycle, the 2's complement values are obtained when the operands are negative. In the second and third cycles, each 264-bit operand is divided into two 132-bit segments, and 132-bit $\times$ 132-bit multiplications in parallel generate partial products. For

these multiplications, embedded multipliers in the target device, DSP48E2 [36], which perform 27-bit × 18-bit multiplication, are allocated through logic synthesis in the FPGA design tool Vivado [37]. Finally, in the fourth cycle, the summation of the partial products as well as the 2's complement operation, if necessary, yields a multiplication result. Simple ∗ and + operators are used in the HDL source files for partial multiplication and addition, respectively.

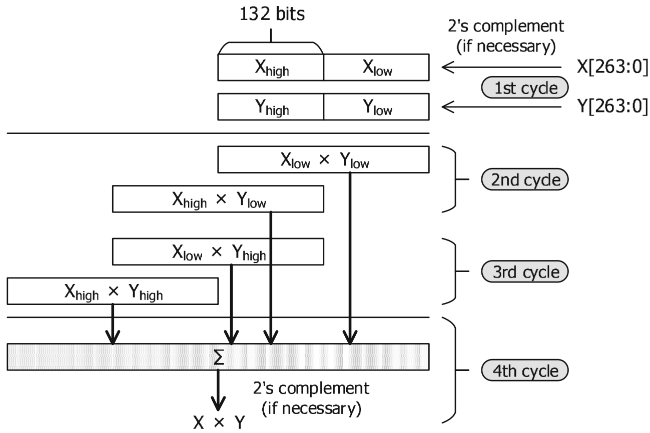

Fig. 1. 264-bit × 264-bit multiplier.

Multi-bit Modular Montgomery Multiplier. The algorithm of our modular multiplication is based on the Montgomery reduction technique. The designed Montgomery multiplier receives two 260-bit input data, x_1 and x_2, and generates a 260-bit result, z, through two 520-bit additions, two 264-bit subtractions, and six 264-bit × 264-bit multiplications. These calculations are performed serially since no parallelism is inherent in the Montgomery multiplication algorithm. The dataflow for Montgomery multiplication is shown in Fig. 2. In Steps 1, 2, 3, 6, 7, and 8, the above-mentioned multi-bit multiplier is used. Each multiplication process takes four cycles. In Steps 4 and 9, a multi-bit adder is used in one cycle. Subtraction is performed for residue calculation in Steps 5 and 10. A total of 28 cycles are used for processing.

Multi-bit Modular Inversion Calculator. An inversion calculator is designed such that it inputs a 260-bit data and outputs a 256-bit result that is the corresponding inverse element over a prime field. The calculation is based on the binary extended GCD algorithm [28], wherein one subtraction or two subtractions in parallel along with one-bit shift operations (insignificant delay) are conditionally performed in each loop iteration. Considering that the operands are 260-bit data and that, in contrast, 520-bit additions are performed in one

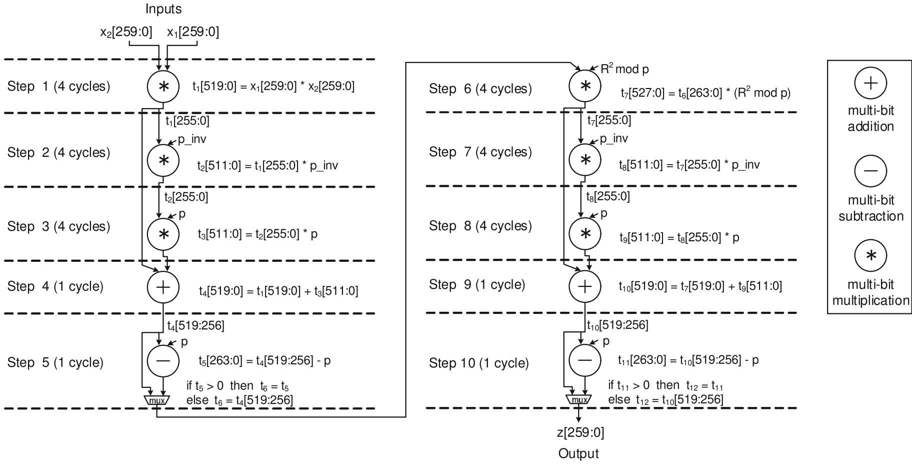

Fig. 2. Dataflow in Montgomery multiplication.

clock cycle in the Montgomery multiplier, our strategy is to unroll two consecutive iterations so that two subtractions are serially performed in one clock cycle. As a result, it executes half the number of iterations compared with the original algorithm and spends half of the clock cycles.

Figure 3(a) shows the pseudocode of the original binary extended GCD algorithm. The variables u, v, B, and D are used in the computation. According to their values, shift, subtraction, or subtraction and shift operations are applied. The shift operations are performed without combinational logic in the hardware implementation, as they are achieved simply by connecting wires appropriately, whereas the subtractions are performed by subtractors. In contrast, in our inversion algorithm, shown in Fig. 3(b), two consecutive iterations in the original algorithm are unrolled and executed in an iteration. Additional variables—$u0$, $v0$, $B0$, and $D0$—hold the temporal results of the first part that are then used in the second part. This unrolling technique causes two subtractions, at most, to be cascaded in an iteration. For example, two subtractions in lines (A) and (B) in Fig. 3(b) are serially executed, when u, v, $u0$, and $v0$ are odd, $u \geq v$, and $u0 \geq v0$. These cascaded subtractions are within one clock-cycle delay in our implementation. In addition, six subtractors are implemented and reused, whereas 12 subtractions are to be executed in the algorithm.

The number of iterations in the original binary extended GCD algorithm depends on the input data. As a measure against side-channel attacks, dummy iterations are added after completing the calculation until it reaches a predefined upper bound, so that the execution time is fixed regardless of the input data. In addition, random calculations are performed in dummy iterations to maintain the energy dissipation. As the predefined upper bound, we use 742 for a 256-bit prime field, which is the theoretical upper bound for the binary extended GCD algorithm [3]. This leads to an execution time of $742/2 = 371$ clock cycles for the main loop in the algorithm.

```
# Initialization
u <= p;   # field modulus
v <= x;   # input data
B <= 0;  D <= 1;

# Main loop
while ( u > 0 ) do
  if ( u is even ) then
    u <= u >> 1;   # 1-bit right shift
    if ( B is even ) then
      B <= B >> 1;
    else
      B <= (B – p) >> 1;
    end if;
    v <= v;  D <= D;
  elsif ( v is even ) then
    v <= v >> 1;
    if ( D is even ) then
      D <= D >> 1;
    else
      D <= (D – p) >> 1;
    end if;
    u <= u;  B <= B;
  elsif ( u >= v ) then
    u <= u – v;  B <= B – D;
    v <= v;      D <= D;
  else
    v <= v – u;  D <= D – B;
    u <= u;      B <= B;
  end if;

end while;   # D is final output
```

(a) Original inversion algorithm

```
# Initialization
u <= p;   # field modulus
v <= x;   # input data
B <= 0;  D <= 1;

# Main loop
while ( u > 0 ) do
# Corresponding to 1st iteration
  if ( u is even ) then
    u0 <= u >> 1;   # 1-bit right shift
    if ( B is even ) then
      B0 <= B >> 1;
    else
      B0 <= (B – p) >> 1;
    end if;
    v0 <= v;  D0 <= D;
  elsif ( v is even ) then
    v0 <= v >> 1;
    if ( D is even ) then
      D0 <= D >> 1;
    else
      D0 <= (D – p) >> 1;
    end if;
    u0 <= u;  B0 <= B;
  elsif ( u >= v ) then
    u0 <= u – v;  B0 <= B – D; • • • • (A)
    v0 <= v;      D0 <= D;
  else
    v0 <= v – u;  D0 <= D – B;
    u0 <= u;      B0 <= B;
  end if;

# Corresponding to 2nd iteration
  if ( u0 = 0 ) then
    u <= u0;  v <= v0;  B <= B0;  D <= D0;
  else   # u0 > 0
    if ( u0 is even ) then
      u <= u0 >> 1;
      if ( B0 is even ) then
        B <= B0 >> 1;
      else
        B <= (B0 – p) >> 1;
      end if;
      v <= v0;  D <= D0;
    elsif ( v0 is even ) then
      v <= v0 >> 1;
      if ( D0 is even ) then
        D <= D0 >> 1;
      else
        D <= (D0 – p) >> 1;
      end if;
      u <= u0;  B <= B0;
    elsif ( u0 >= v0 ) then
      u <= u0 – v0;  B <= B0 – D0; • • • • (B)
      v <= v0;       D <= D0;
    else
      v <= v0 – u0;  D <= D0 – B0;
      u <= u0;       B <= B0;
    end if;
  end if;

end while;   # D is final output
```

(b) Revised algorithm with unrolling

Fig. 3. Inversion algorithms.

5.2 Design of EC Point Computation Modules and Parallelism

Based on the directions mentioned in the previous section, we designed and implemented the following EC point computation modules: affine point addition module (PADD), affine point doubling module (PDBL), extended affine point addition module (EXA_PADD), and extended affine point doubling module (EXA_PDBL).

EC Point Computation Modules. PADD, PDBL, EXA_PADD, and EXA_PDBL modules are composed of the aforementioned arithmetic units (adders/subtractors, Montgomery multipliers, and inversion calculators) with predefined control sequences. The data are transmitted between the arithmetic units via 260-bit temporary registers. To achieve parallel processing between the EC point computation modules, as described later, the designed system is equipped with six temporary registers: one of them is occupied by the PADD module, the other two are shared by the PADD and EXA_PADD modules, and the remaining three are shared by the PDBL, EXA_PDBL, and EXA_PADD modules.

Figures 4(a) and (b) show the dataflow in PADD and PDBL, respectively, and Figs. 5(a) and (b) show the dataflow in EXA_PADD and EXA_PDBL, respectively. Parallel processing contributes not only to high-performance processing, but also to resistance to SPA, as it makes power analysis more difficult than in sequential processing. The system utilizes two types of parallelism: intra-module and inter-module parallelism. In terms of the former, parallelism inside

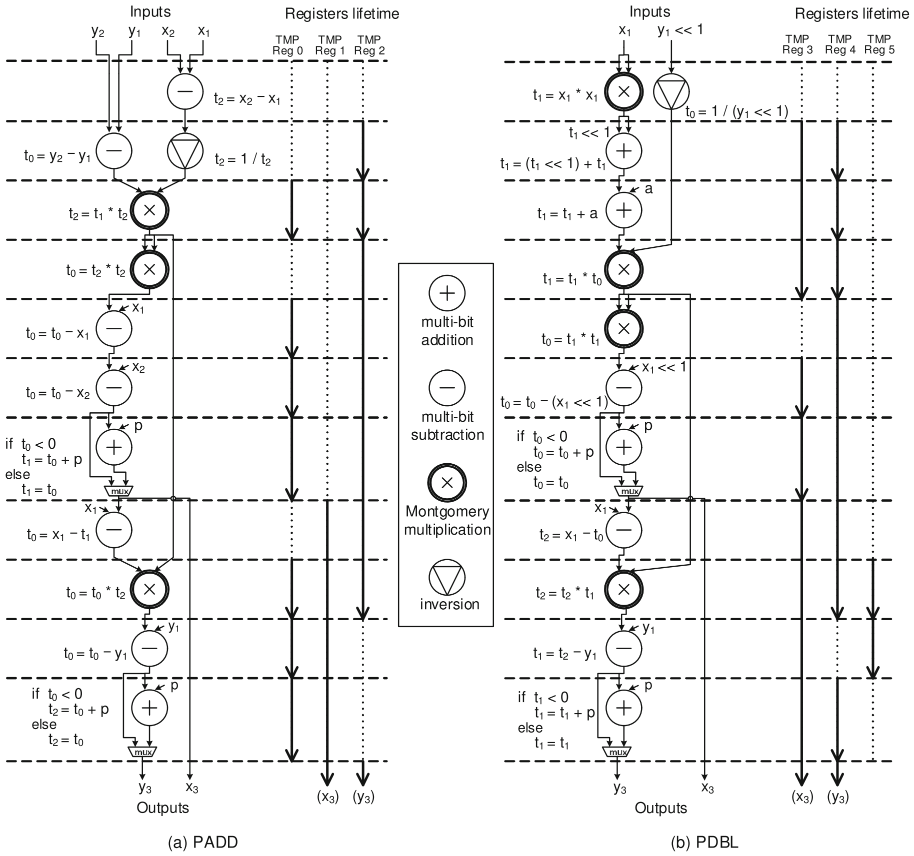

Fig. 4. Dataflow in (a) PADD and (b) PDBL.

EXA_PADD is described as an example. Analyzing the dataflow and considering the processing time of each arithmetic unit offers the possibility of parallel execution among arithmetic. In Fig. 5(a), during the inversion processing, addition, subtraction, and three Montgomery multiplications can be processed and completed. The same strategy is applied to the other EC point computation modules (PADD, PDBL, and EXA_PDBL), although the amount of parallelism extracted is small for them. After scheduling the arithmetic units and allocating registers, the number of necessary temporary registers is known. That is, three registers are necessary for PADD, PDBL, and EXA_PDBL, whereas five registers are necessary for EXA_PADD.

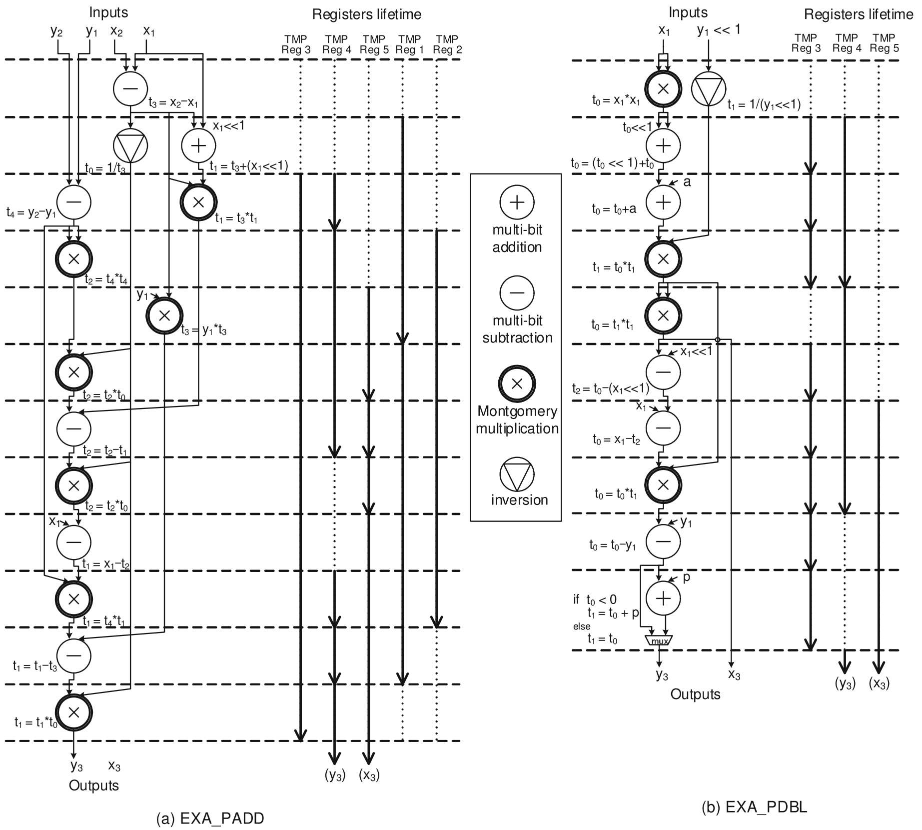

Fig. 5. Dataflow in (a) EXA_PADD and (b) EXA_PDBL.

After intra-module parallelism is fixed, inter-module parallelism is established. In the main loop of Algorithm 1, an affine point addition (Step 6) and affine point doubling (Step 7) are executed. The latter does not have read-after-write dependency with the former. Therefore, affine point doubling can be executed in parallel with the preceding affine point addition. Affine point doubling has a write-after-read relation with the affine point addition in terms of A, that is, update of A by affine point doubling has to be performed after the affine point addition reads it. This is solved by introducing synchronization mechanisms, as described in the next subsection. In addition to this parallelism, at Step 10 in the final correction, affine point addition and extended affine point doubling are processed in parallel. Considering the inter-module parallel processing and the number of temporary registers required by each EC point computation module, the temporary registers are efficiently shared between modules as mentioned above. As a result, the introduction of the two extended affine point computations does not require additional registers. Thus, enhancing security for exceptional addition can be done with no additional registers.

5.3 APIs for Inter-module Parallelism and Synchronization

To achieve parallel processing and synchronization between EC point computation modules, the software procedures shown in Table 1 are implemented. These APIs enable our efficient hardware–software co-processing.

For each EC point computation module, the corresponding Start_*() procedure invokes the hardware module and finishes (or returns to the caller) asynchronously, i.e., without waiting for the completion of the hardware operation. In contrast, when End_*() is called, it waits for the completion of the corresponding hardware processing. As affine point doubling has a write-after-read relation with the affine point addition in the main loop, its result is not directly written in the corresponding buffers (described later) but written in the temporary registers. The execution of Sync_PDBL() moves the result to the target buffer.

Using these procedures, the main loop of Algorithm 1 is written in C language, as in Fig. 6. With the use of the APIs described above, PDBL and PADD run in parallel. Here, Start_*() procedures have parameters for EC point data. For Start_PDBL(), the buffer identifier "2" is specified since Buffer[2] contains an EC point A. Similarly, for Start_PADD(), the buffer identifiers "2" and "ki" are specified so that Buffer[ki] is updated by Buffer[2] + Buffer[ki]. In contrast, Sync_PDBL() is accompanied by the destination buffer identifier "2" so that the temporary register value is copied to Buffer[2].

In addition to the parallelism between affine point doubling and affine point addition, affine point addition and extended affine point doubling can run in parallel in Step 10 of Algorithm 1. Considering these chances of parallelism, resource sharing for temporary registers and arithmetic units is performed. The possible combinations of EC point computations for parallel processing are a pair of PADD and PDBL and a pair of PADD and EXA_PDBL. Each EC point computation module uses a set of an adder, a subtractor, a Montgomery multiplier, and an inversion calculator. To reduce the total hardware amount, a set of arithmetic units is shared between PADD and EXA_PADD and another set is

Table 1. Application Programming Interfaces (APIs).

Procedure	Action
Start_PADD()	Invokes PADD
End_PADD()	Waits for completion of PADD
Start_PDBL()	Invokes PDBL
End_PDBL()	Waits for completion of PDBL
Sync_PDBL()	Stores the result of PDBL in buffers
Start_EXA_PADD()	Invokes EXA_PADD
End_EXA_PADD()	Waits for completion of EXA_PADD
Start_EXA_PDBL()	Invokes EXA_PDBL
End_EXA_PDBL()	Waits for completion of EXA_PDBL

```
for ( i = 1; i < l; i ++ ) {
    Start_PDBL( 2 );      /* TMP_Reg = 2 * Buffer[2] */
    ki = (k >> i) & 0x1;
    Start_PADD( 2, ki );  /* Buffer[ki] = Buffer[2] + Buffer[ki]; */
    End_PADD();
    End_PDBL()
    Sync_PDBL( 2 );       /* writing TMP_Reg in Buffer[2] */
}
```

Fig. 6. Software code of main loop in Algorithm 1.

shared between PDBL and EXA_PDBL (Fig. 7). Similarly, temporary registers TMP-Reg 1 and TMP-Reg 2 can be shared between PADD and EXA_PADD, and TMP-Regs 3 to 5 are shared between PDBL, EXA_PDBL, and EXA_PADD.

5.4 System Structure

The designed ECC system was implemented in Xilinx Zynq UltraScale+ MPSoC ZU7EV device [39]. Figure 7 depicts the system structure, including the designed arithmetic units and EC point computation modules. A Cortex-A53 core in the processing system (PS) executes software code at a clock frequency of 500 MHz.

EC point computation modules with arithmetic units are implemented in programmable logic (PL), working at a clock frequency of 214.286 MHz[1]. In the figure, the multi-bit adder, subtractor, multiplier, Modular Montgomery multiplier, and inversion calculator are depicted as ADD, SUB, MUL, MONT_MUL, and INVERSE, respectively. Each EC point computation module includes a control register and a status register. PS software calls Start_*() procedure, which writes an invocation signal as well as buffer identifiers in the control register. Similarly, it calls End_*() and reads from the status register to recognize the completion of module processing. The control and status registers are memory-address-mapped and accessed via conventional load/store instructions[2].

Data transfer between PS and PL is performed through a high-speed on-chip bus (AXI), and the unit of transfer is 64 bits. EC point data are transferred via four global buffers (Buffer[0–3] in the figure). These buffers are memory-address-mapped and accessed by conventional load/store instructions. Each buffer contains point data (256 bits $\times$ 2) on the elliptic curve. Each EC point computation module uses buffers specified by the control register. During computation, the temporary registers (TMP-Regs 0 to 5 in the figure) are used to store the results of the arithmetic units.

[1] Phase Locked Loop (PPL) in the device generates 214.286 MHz by 33.3 MHz $\times$ 90/14.

[2] As ARM processors use relaxed memory models, memory barrier (DMB) instructions must be properly inserted to guarantee access order to the control and status registers.

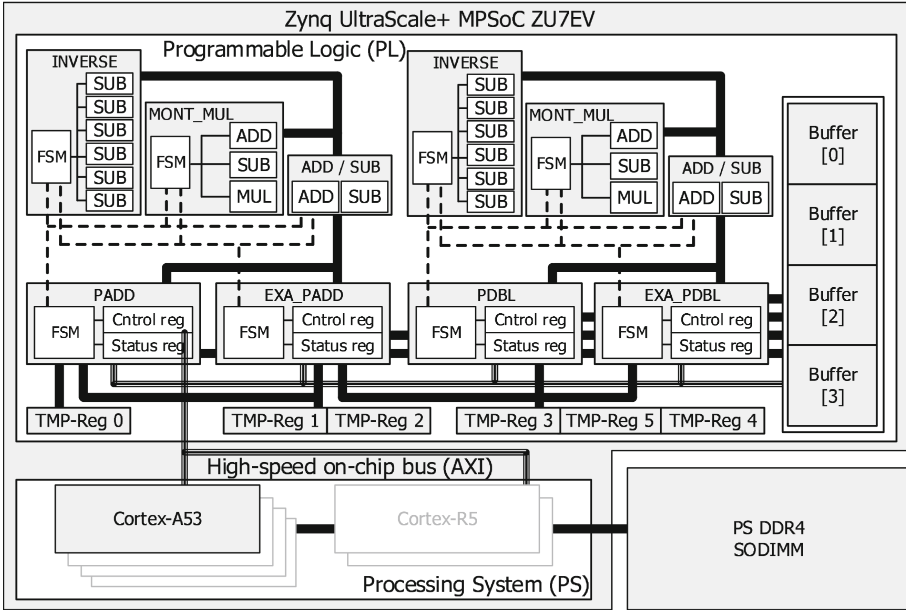

Fig. 7. System structure for Algorithm 1.

5.5 Execution Cycles

Table 2 shows the execution clock cycles of the arithmetic units, EC point computation modules, and scalar multiplication. Our design gives constant execution times for all the computation modules. PADD and PDBL take almost the same number of cycles, leading to balanced parallel processing in the main loop.

The execution cycles for scalar multiplication are 120 403, corresponding to 0.562 ms at the clock frequency in our implementation, that is, 214.286 MHz. These cycles do not include software execution, such as API procedures and operands transfer. The total execution time of the hardware–software co-processing is described in the next section.

6 Analysis

This section presents the evaluation results in terms of performance and hardware-resource usage. Table 3 compares our designs with the other existing FPGA designs for generic ECC over 256-bit prime fields described in Sect. 3. As the FPGA devices used are different in terms of their generations, the table includes, in the right-most column, the processing time normalized to 200-MHz processing for reference. It also compares them from the point of view of security of exception-free for any k and usage of pre-computation tables.

Table 2. Execution clock cycles of arithmetic units, EC point computation modules, and scalar multiplication.

Arithmetic unit	Cycles
ADD/SUB	1
MUL	4
MONT_MUL	28
INVERSE	372

EC point computation module	Cycles
PADD	463
PDBL	461
EXA_PADD	488
EXA_PDBL	460

Scalar multiplication	Cycles
PDBL×1 + PADD×258 + EXA_PADD×1	120,403

6.1 Execution Time

The designed ECC system was synthesized and implemented with Xilinx Vivado v2019.2. The processing time of the ECC scalar multiplication was measured on a ZCU104 evaluation board [38]. Software with Linux 4.14.0 runs on the processor (Cortex-A53) in PS at 500 MHz. The software code is written in C language and compiled using gcc 6.3.0 with -O4 option. The elapsed time was obtained using the gettimeofday() library function. The elapsed time includes not only the hardware processing time but also the software processing time.

For comparison, software-only processing, “Soft,” that executes Algorithm 1 using GNU Multiple Precision Arithmetic Library (GMP) Version 6.1.2 [1] is prepared. Our proposed system of processing with hardware modules is “w/HW.” Another implementation is “w/HW-auto,” equipped with an auto loop mechanism, where the main loop sequence is automatically processed in the hardware (without Start/End_PADD/PDBL()) to mitigate the overhead of PS-from/to-PL communication/synchronization.

For our implementations (Soft, w/HW, and w/HW-auto), the average processing time of scalar multiplication for 1000 pairs of $(k,\ P)$ is presented in Table 3. Soft takes 7.943 ms, which is not fast enough for various real-time applications. In contrast, the execution time of w/HW is 0.742 ms, which is approximately 11 times faster than Soft. In addition, w/HW-auto takes 0.575 ms, which is 23% faster than w/HW and 14 times faster than Soft. This result implies that the overhead of invoking hardware and recognizing its completion, i.e., writing to/reading from control/status registers, is non-negligible.

Table 3 shows that our design is the fastest among the existing FPGA implementations without a precomputation table. Let us compare our design with [24], which uses 15 points for a pre-computation table. Thanks to the pre-computation table, it can reduce the number of point additions to 71 from the original 256. Nevertheless, our design is comparable to it, although processing executes 256 point additions and doublings without pre-computation table, each of which involves inversion calculation. This indicates that the number of clock cycles to be taken by our design is sufficiently low. In other words, our design achieves

Table 3. Comparison of scalar multiplication over arbitrary 256-bit prime fields.

Design	Exception free (any k)	Pre-comp. Table (# points)	Device	Area	Frequency (MHz)	Time (ms)	Time at 200 MHz (ms)
Soft (GMP)	Yes	No	Cortex-A53	N/A	500	7.943	–
This work							
w/HW w/HW-auto	Yes	No	Zynq UltraScale+	6.3K slices (42K LUTs) + 256 DSPs	214.286	0.742 0.575	0.795 0.626
[22] (2019)	No	No	Virtex-7	5.4K slices	124.2	3.730	2.316
[13] (2018)	No	No	Virtex-4	9.4K slices	20.44	29.840	3.050
[12]	No	No	Kintex-7	11.3K slices	121.5	3.270	1.987
[14] (2016)	No	No	Virtex-6 Virtex-4	(No report) 1.3K slices	70 40	2.800 5.000	0.980 1.000
[7] (2011)	No	No	Virtex-4 Virtex-II Pro	(no report) 12K slices (20K LUTs)	54 36	6.260 9.380	1.690 1.688
[6] (2009)	No	No	Virtex-4	20K slices (34K LUTs)	43.32	7.700	1.668
[24] (2013)	Yes	Yes (15 pts)	Virtex-5	1.7K slices (4.2K LUTs) + 37 DSPs	291	0.380	0.553
[8] (2010)	No	No	Stratix II	9K ALMs + 96 DSPs	157.2	0.680	0.534

more efficient processing per cycle. Therefore, we conclude that our ECC system should be based on a highly efficient digital logic design.

The performance gain of our designs is attributed to the high performance in inverse calculation and utilization of intra-/inter-module parallelism. Table 4 shows the performances of several high-performance designs for modular inversion over 256-bit prime fields. All implementations, including our design, in the table are based on the extended Euclidean algorithm or its variants. This means that the processing time of these implementations depends on the input values. Their processing times in Table 4 are from the corresponding literature, which are regarded as the average execution times. In contrast, our implementation of inversion, “Ours w/ UB” in the table, shows the execution time of inversion with the upper bound mentioned in Sect. 5.1, which is the fixed execution time, whereas “Ours w/o UB”, which is for reference, corresponds to the average execution time when the upper-bounded loop execution is not applied. The table shows that our designed inversion calculator is the fastest among the recently published designs.

Table 4. Performance of modular inversion over 256-bit prime fields.

Design	Device	Area	Frequency (MHz)	Time (μs)	Time at 200 MHz (μs)
Ours w/o UB	Zynq	6,926 LUTs	214.286	1.279	1.370
Ours w/UB	UltraScale+			1.736	1.860
[21] (2019)	Virtex-7	1,069 slices	168.560	2.013	1.697
[5] (2018)	Virtex-7	617 slices	144.011	2.220	1.599
[11] (2015)	Virtex-7	1,480 slices	146.380	2.329	1.705
[15] (2016)	Virtex-6	4,758 LUTs	151.000	3.391	2.560

6.2 FPGA Resources

Table 3 and Table 4 include the information of FPGA resources (Area) occupied by each design. As the FPGA devices used are different from design to design, directly comparing their sizes is difficult. For example, the UltraScale+ architecture has a slice structure containing eight 6-input look-up tables (LUTs), whereas a slice in Virtex-6/7 has four 6-input LUTs, or Virtex-4 has 4-input LUTs. Nevertheless, the proposed system seems to occupy more resources than the others. However, the size is sufficiently practical since the area information of our implementation reported in Table 3 is not only for scalar multiplication processing but also for all other components including the high-speed on-chip bus and the DDR4 DIMM controller, and the total hardware can be accommodated using low-price FPGA devices such as a Xilinx Artix-7 XC7A200T that comprises 134 600 LUTS and 740 DSPs [35].

7 Conclusion

We have investigated various methods for efficient FPGA implementations of scalar multiplication on elliptic curve cryptosystems over any prime field. Our design makes the most of the advanced and high-performance programmable logic in today's FPGA devices and extracts the full parallelism inherent in the algorithms. Our proposed hardware–software coprocessing outperforms the existing FPGA implementations for generic ECC over 256-bit prime fields without a pre-computation table and is also secure against SPA and exception-free for any scalar. The processing time result of 0.575 ms shows that our design could be applicable to any real-time embedded system.

Acknowledgments. This work was supported by enPiT (Education Network for Practical Information Technologies) at MEXT, Innovation Platform for Society 5.0 at MEXT, and JSPS KAKENHI Grant Number JP21H03443.

References

1. https://gmplib.org/
2. 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)
3. Bernstein, D.J., Yang, B.-Y.: Fast constant-time GCD computation and modular inversion. IACR Trans. Cryptogr. Hardw. Embedded Syst. **2019**(3), 340–398 (2019)
4. Blake, I., Seroussi, G., Smart, N.: Elliptic Curves in Cryptography. Cambridge University Press, Cambridge (1999)
5. Dong, X., Zhang, L., Gao, X.: An efficient FPGA implementation of ECC modular inversion over F'_{256}. In: Proceedings International Conference on Cryptography, Security and Privacy, pp. 29–33 (2018)
6. Ghosh, S., Alam, M., Chowdhury, D.R., Guputa, I.S.: Parallel crypto-devices for GF(p) elliptic curve multiplication resistant against side channel attacks. Comput. Electr. Eng. **35**(2), 329–338 (2009)
7. Ghosh, S., Mukhopadhyay, D., Roychowdhury, D.: Petrel: power and timing attack resistant elliptic curve scalar multiplier based on programmable GF(p) arithmetic unit. IEEE Trans. Circ. Syst. **58**(8), 1798–1812 (2011)
8. Guillermin, N.: A high speed coprocessor for elliptic curve scalar multiplications over $\mathbb{F}_p$. In: Proceedings of International Conference on Cryptographic Hardware and Embedded Systems, pp. 48–64 (2010)
9. 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). https://doi.org/10.1007/978-3-540-85053-3_5
10. Hankerson, D., Menezes, A.J., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer, New York (2004). https://doi.org/10.1007/b97644
11. Hossain, M.S., Kong, Y.: High-performance FPGA implementation of modular inversion over $\mathbb{F}_{256}$ for elliptic curve cryptography. In: Proceedings of IEEE International Conference on Data Science and Data Intensive Systems, pp. 169–174 (2015)
12. Hossain, M.S., Kong, Y., Saeedi, E., Vayalil, N.C.: High-performance elliptic curve cryptography processor over NIST prime fields. IET Comput. Digit. Tech. **11**(1), 33–42 (2017)
13. Hu, X., Zheng, X., Zhang, S., Cai, S., Xiong, X.: A low hardware consumption elliptic curve cryptographic architecture over GF(p) in embedded application. Electronics **7**(7), 13p (2018)
14. Javeed, K., Wang, X.: FPGA based high speed SPA resistant elliptic curve scalar multiplier architecture. Int. J. Reconfig. Comput. **2016**(5), 1–10 (2016)
15. Javeed, K., Wang, X.: Low latency flexible FPGA implementation of point multiplication on elliptic curves over GF(p). Int. J. Circuit Theory Appl. **45**(2), 214–228 (2016)
16. Jin, Y., Miyaji, A.: Secure and compact elliptic curve cryptosystems. In: Jang-Jaccard, J., Guo, F. (eds.) ACISP 2019. LNCS, vol. 11547, pp. 639–650. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21548-4_36
17. Joye, M.: Highly regular m-Ary powering ladders. In: Jacobson, M.J., Rijmen, V., Safavi-Naini, R. (eds.) SAC 2009. LNCS, vol. 5867, pp. 350–363. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05445-7_22

18. Karatsuba, A.A., Ofman, Y.: Multiplication of multidigit numbers on automata. Soviet Phys. Doklady **7**(7), 595–596 (1963)
19. Koblitz, N.: Elliptic curve cryptosystems. Math. Comput. **48**, 203–209 (1987)
20. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-68697-5_9
21. Kudithi, T., Sakthivel, R.: An efficient hardware implementation of finite field inversion for elliptic curve cryptography. Int. J. Innov. Technol. Explor. Eng. **8**(9), 827–932 (2019)
22. Kudithi, T., Sakthivel, R.: High-performance ECC processor architecture design for IoT security applications. J. Supercomput. **75**(1), 447–474 (2019). https://doi.org/10.1007/s11227-018-02740-2
23. Le, D.-P., Nguyen, B.P.: Fast point quadrupling on elliptic curves. In: Proceedings of Symposium on Information and Communication Technology, pp. 218–222 (2012)
24. Ma, Y., Liu, Z., Pan, W., Jing, J.: A high-speed elliptic curve cryptographic processor for generic curves over GF(p). In: Proceedings of International Conference on Selected Areas in Cryptography, pp. 421–437 (2013)
25. Mamiya, H., Miyaji, A., Morimoto, H.: Secure elliptic curve exponentiation against RPA, ZRA, DPA, and SPA. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **89-A**(8):2207–2215 (2006)
26. Marzouqi, H., Al-Qutayri, M., Salah, K., Saleh, H.: A 65 nm ASIC based 256 NIST prime field ECC processor. In: Proceedings of IEEE 59th International Midwest Symposium on Circuits and Systems, pp. 1–4 (2016)
27. Marzouqi, H., Al-Qutayri, M., Salah, K., Schinianakis, D., Stouraitis, T.: A high-speed FPGA implementation of an RSD-based ECC processor. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **24**(1), 151–164 (2016)
28. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996)
29. 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). https://doi.org/10.1007/3-540-39799-X_31
30. Möller, B.: Parallelizable elliptic curve point multiplication method with resistance against side-channel attacks. In: Chan, A.H., Gligor, V. (eds.) ISC 2002. LNCS, vol. 2433, pp. 402–413. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45811-5_31
31. Montgomery, P.L.: Modular multiplication without trial division. Math. Comput. **44**(170), 519–521 (1985)
32. 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). https://doi.org/10.1007/978-3-662-49890-3_16
33. Shylashree, N., Sridhar, V., Patawardhan, D.: FPGA based efficient elliptic curve cryptosystem processor for NIST 256 prime field. In: Proceedings of IEEE Region 10 Conference, pp. 194–199 (2016)
34. Wu, X., Chouliaras, V., Goodall, R.: An application-specific processor hard macro for real-time control. In: Proceedings of IEEE International SOC Conference, pp. 369–372 (2004)

35. Xilinx, Inc.: 7 Series FPGAs Data Sheet: Overview, DS180 (v2.6)
36. Xilinx, Inc.: UltraScale Architecture DSP Slice User Guide, UG579 (v1.10)
37. Xilinx, Inc.: Vivado Design Suite User Guide, Synthesis UG901 (v2020.1)
38. Xilinx, Inc.: ZCU104 Evaluation Board User Guide, UG1267 (v1.1)
39. Xilinx, Inc.: Zynq UltraScale+ MPSoC Data Sheet: Overview, DS891 (v1.8)