computers

MDPI

Article

Area–Time-Efficient High-Radix Modular Inversion Algorithm and Hardware Implementation for ECC over Prime Fields

Yamin Li

Computer Architecture Laboratory, Department of Computer Science, Faculty of Computer and Information Sciences, Hosei University, Tokyo 184-8584, Japan; yamin@hosei.ac.jp

Abstract: Elliptic curve cryptography (ECC) is widely used for secure communications, because it can provide the same level of security as RSA with a much smaller key size. In constrained environments, it is important to consider efficiency, in terms of execution time and hardware costs. Modular inversion is a key time-consuming calculation used in ECC. Its hardware implementation requires extensive hardware resources, such as lookup tables and registers. We investigate the state-of-the-art modular inversion algorithms, and evaluate the performance and cost of the algorithms and their hardware implementations. We then propose a high-radix modular inversion algorithm aimed at reducing the execution time and hardware costs. We present a detailed radix-8 hardware implementation based on 256-bit primes in Verilog HDL and compare its cost performance to other implementations. Our implementation on the Altera Cyclone V FPGA chip used 1227 ALMs (adaptive logic modules) and 1037 registers. The modular inversion calculation took 3.67 ms. The AT (area–time) factor was 8.30, outperforming the other implementations. We also present an implementation of ECC using the proposed radix-8 modular inversion algorithm. The implementation results also showed that our modular inversion algorithm was more efficient in area–time than the other algorithms.

Keywords: computer security; elliptic curve cryptography; modular inversion; hardware; Verilog HDL; FPGA; cost performance evaluation

Citation: Li, Y. Area–Time-Efficient High-Radix Modular Inversion Algorithm and Hardware Implementation for ECC over Prime Fields. *Computers* **2024**, *13*, 265. https://doi.org/10.3390/computers13100265

Academic Editors: Helge Janicke and Leandros Maglaras

Received: 10 September 2024
Revised: 8 October 2024
Accepted: 10 October 2024
Published: 12 October 2024

1. Introduction

Nowadays, Internet of Things (IoT) applications use hardware security modules to ensure secure communications. In such a constrained environment, execution time and hardware costs are critical to efficient system design. Elliptic curve cryptography (ECC) is one of the most advanced public key cryptographic techniques. It requires a smaller key than other methods, to achieve roughly the same level of security.

ECC can be used to provide secure key agreement between two parties over an insecure network. It can also be used for digital signatures, to verify the authenticity and integrity of digital messages. Modular inversion is a critical operation in ECC. ECC calculates points on an elliptic curve over a finite field (such as a field of prime numbers) based on point addition (PA) and point doubling (PD) computations. In affine coordinates, PA and PD must calculate the slope of a line. Such calculations involve costly modular inversions. In projective or Jacobian coordinates, PA and PD do not require such calculations, but a modular inversion is still required to transform the points into affine coordinates to obtain the same key for the two parties.

Given a prime number m, the inverse r of a number a with $a < m$ is defined as $r = a^{-1} \bmod m$. There are two main popular methods for calculating modular inversion:

1. Extended Euclidean algorithm (EEA) without using divisions.
2. Using Fermat's little theorem $a^{m-1} = 1 \bmod m$ [1]: $r = a^{m-2} \bmod m = a^{-1} \bmod m$.

We will see that the method using Fermat's little theorem takes more time and requires more registers than the EEA. Therefore, we focus our design on the use of the EEA.

The EEA inherently needs divisions. It calculates the integer quotient and the remainder based on the quotient. The divisions can be replaced by addition, subtraction, and shift operations. For simplicity, we will also refer to the EEA which does not use divisions as an EEA.

To calculate $r = a^{-1} \bmod m$, the EEA first initializes the variables u, v, x, y with inputs $a, m, 1, 0$, respectively. Then, the EEA repeats calculations containing only addition, subtraction, and shift operations on u, v, x, y until $u = 1$ or $v = 1$. Finally, the modular inversion result is available by adjusting x or y, corresponding to $u = 1$ or $v = 1$. A modular inversion algorithm is said to be fast if u or v reaches 1 quickly.

The most widely used modular inversion algorithm is Algorithm 2.22, proposed by Hankerson, Menezes, and Vanstone [2]. It repeatedly shifts u or v to the right when u or v is even. Correspondingly, x is also shifted to the right with the shift of u; y is also shifted to the right with the shift of v. Note that, when x or y is odd, m will be added before the shift. This guarantees that the value to be right-shifted is even, since the prime m is odd. Next, if $u \geq v$, u and x are replaced by $u - v$ and $x - y$, respectively. Otherwise, v and y are replaced by $v - u$ and $y - x$, respectively. Finally, the result is $x \bmod m$ if $u = 1$, and $y \bmod m$ otherwise. Hossain and Kong [3] revised Algorithm 2.22 by adding m to x or y if it is negative. This ensures that x and y are non-negative. Daly, Marnane, Kerins, and Popovici [4] revised Algorithm 2.22 by dividing $u - v$ or $v - u$ by two because the subtraction result is even (both u and v are odd before the subtraction). Correspondingly, $x - y$ or $y - x$ also needs to be divided by two: If $x - y$ or $y - x$ is odd, m is added before the division. Division by two is performed by shifting one bit to the right. Mrabet, El-Mrabet, Bouallegue, Mesnager, and Machhout proposed a modular inversion algorithm [5] with $u + v$. Instead of $u - v$ or $v - u$, as Algorithm 2.22 uses, they perform $u + v$ for new u or v. This operation slows down the speed at which u or v reaches 1, increasing the execution time. Chen and Qin proposed a modular inversion algorithm [6] that only uses adders. Subtractions are performed by addition with inversion and addition by 1. Choi, Lee, Kong, and Kim proposed a modular inversion algorithm [7] that replaces the repeated shift of u or v and the corresponding shift of x or y in Algorithm 2.22 by a selection of u, x or $0, 0$, or a selection of v, y or $0, 0$, based on the even/odd of v or u. This simplifies the circuit by replacing adders with multiplexers, reducing the circuit delay. In addition, they use $-v$ and $-y$, instead of v and y, during the calculation. This merges $u - v$ and $v - u$ into $u + v$ and merges $x - y$ and $y - x$ into $x + y$, reducing the circuit cost. Mixed radix-4 modular inversion algorithms were investigated in [7–10]. If u or v is divisible by four, u or v is shifted to the right by two bits. Otherwise, if u or v is even (divisible by two), u or v is shifted to the right by one bit. Otherwise (both u and v are odd), $u - v$ or $v - u$ is shifted to the right by one bit and assigned to u or v. Correspondingly, x or y is adjusted by adding $-m$, m, or $2m$ and shifted to the right by two bits or one bit. [8] proposed a radix-4 modular inversion algorithm that uses sequential condition checking for the calculation of u, v, x, and y. [9] implemented the SM2 ECC protocol. The iterations of the modular inversion are controlled by the bit counter ρ, resulting in unnecessary iterations. Using u and v to control the iterations will finish the calculation quickly. [10] presented a radix-4 version of Algorithm 2.22. Dong, Zhang, and Gao proposed a mixed radix-8 modular inversion algorithm [11] that uses extensive hardware resources. Hao et al. presented a lightweight architecture for elliptic curve scalar multiplication over prime fields [12]. They revised Algorithm 2.22 by using only adders and forced $x - y$ and $y - x$ to be in the range 0 to m. Guo et al. proposed a modular inversion algorithm [13] that makes v always be odd. If u is even, it is shifted one bit to the right. Correspondingly, x is also shifted one bit to the right (if x is odd, m is added before the shift). Otherwise (u is odd), $u - v$ (if $u \geq v$) or $v - u$ (if $u < v$) is shifted one bit to the right, and the shifted value is assigned to u. Meanwhile, $x - y$ or $y - x$ is shifted one bit to the right, and the shifted value is assigned to x. Then, v is updated with u or v; y is updated with x or y. The above calculations are repeated until u becomes 1. Then, the result of the modular inversion is x. In lines 14 and 15 of their algorithm, x_1' and x_2' are compared and x_1 is guaranteed to be non-negative. However, due

to the division by two, the dividend must be adjusted so that it is even. If it is odd, m (p in their algorithm) must be added to it before the division. These codes were not presented in their algorithm.

The AT (area–time) factor is often used for comparisons between implementations. It is defined as the execution time in milliseconds multiplied by the required hardware resources consisting of registers and lookup tables or ALMs (adaptive logic modules).

In this paper, we implement and evaluate all the algorithms mentioned above. We then propose a mixed radix-8 modular inversion algorithm aimed at reducing the execution time and hardware costs. We give a detailed hardware implementation in Verilog HDL based on 256-bit prime numbers. This had lower hardware costs for ALMs and registers and had better performance than the other algorithms. The implementation on the Altera Cyclone V FPGA chip used 1227 ALMs and 1037 registers and took 3.67 ms for the modular inversion computation. It achieved an AT factor of 8.30, lower than all other implementations. We show that the proposed algorithm is also efficient for 192-bit and 521-bit prime numbers. We implemented ECC using different modular inversion algorithms and compared their cost performance. The ECC implementation results showed that our modular inversion algorithm was more efficient in area–time than the other algorithms. We also present an efficient implementation of the Montgomery ladder scalar point multiplication algorithm, a constant execution time algorithm that is resistant to side-channel attacks. The short execution time and low hardware cost of our algorithm and implementation are significant advantages, especially in constrained environments where computing and battery power are limited.

The rest of this paper is organized as follows: Section 2 introduces ECC and modular inversion algorithms. In Section 3, we propose a mixed radix-8 modular inversion algorithm, give its hardware implementation in Verilog HDL, and compare its cost performance with other algorithms. In Section 4, we provide an ECC implementation using the proposed radix-8 modular inversion algorithm and compare its cost performance with ECC implementations using other modular inversion algorithms. This section also presents an efficient implementation of the Montgomery ladder scalar point multiplication algorithm. In Section 5, we discuss some issues related to the algorithm and hardware design. We conclude the paper and suggest some future research topics in Section 6.

2. ECC and Modular Inversion Algorithms

This section introduces ECC and modular inversion algorithms based on the EEA.

2.1. Elliptic Curve Cryptography

ECC [14,15] relies on the fact that scalar point multiplication $Q = dP$ can be computed, but it is almost impossible to compute d given only the original point P and the point of the product Q. An ECC over the finite field of an n-bit prime number m can use the following equation:

$$y^2 = x^3 + ax + b \bmod m \tag{1}$$

For example, the Secp256k1 [16] elliptic curve used in Ethereum Blockchain uses a 256-bit $m = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$. Secp256k1 defines $y^2 = x^3 + ax + b = x^3 + 7$ and gives a point $P = [x, y]$ on the elliptic curve, as follows:

```
a = 0x0000000000000000000000000000000000000000000000000000000000000000
b = 0x0000000000000000000000000000000000000000000000000000000000000007
m = 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f
x = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
y = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
```

The elliptic curve Diffie–Hellman (ECDH) key exchange protocol can be used by two parties, Alice and Bob for example, to establish a shared secret key over an insecure network [16,17]. The ECDH protocol is shown in Table 1.

Table 1. ECDH key exchange.

Expose an Elliptic Curve $y^2 = x^3 + ax + b \bmod m$ and a Point P on the Elliptic Curve to the World	
Alice	**Bob**
Generate a secret d_a Calculate $Q_a = d_a P$ Expose Q_a	Generate a secret d_b Calculate $Q_b = d_b P$ Expose Q_b
Get Q_b from Bob Calculate $Q_{ab} = d_a Q_b$	Get Q_a from Alice Calculate $Q_{ba} = d_b Q_a$
Use x of Q_{ab} as the key	Use x of Q_{ba} as the key

Because $Q_{ab} = d_a Q_b = d_a d_b P$, $Q_{ba} = d_b Q_a = d_b d_a P$, and $d_a d_b = d_b d_a$, we have $Q_{ba} = Q_{ab}$. Below is an ECDH key exchange example using Secp256k1. We can see that the two parties, Alice and Bob, have the same shared secret key (`Qabx` = `Qbax`).

Alice generates and keeps d_a secret and exposes $Q_a = d_a P$:

```
da   = 0x650aa7095daeaa37ab9051541f0ce304f8969a6d88bb3bebb4fe680fca9a2595
Qax  = 0x167d2537aa6bbd8d978b58be0f9466520b7b184e205ff96a9ff567b35b32c7b7
Qay  = 0xde3961553d36551f92726fee0e332133960edddccd2784b98b2af730d2fc6e14
```

Bob generates and keeps d_b secret and exposes $Q_b = d_b P$:

```
db   = 0xedc68f194c4e30d6ef90467df822b00e5ef122dea48c9d1c54817080d1a341f4
Qbx  = 0x839da64a414c2243a5526230603109be9c615613a9e98c3d650bb0488580bbda
Qby  = 0x96e88e99304a5afcdd77c4f3b3327a28162627ebe08194baa0c78dfb67a11042
```

Alice obtains Q_b and calculates $Q_{ab} = d_a Q_b$:

```
Qabx = 0x1f254c7da15899275cdcab9d992f58251a4ab630fe9864d20cf317ab57749947
Qaby = 0xd6cb400b3c49d33d3df28f9d34fa09f8b6c8edf117a378c5a45d0a51e6c0debc
```

Bob obtains Q_a and calculates $Q_{ba} = d_b Q_a$:

```
Qbax = 0x1f254c7da15899275cdcab9d992f58251a4ab630fe9864d20cf317ab57749947
Qbay = 0xd6cb400b3c49d33d3df28f9d34fa09f8b6c8edf117a378c5a45d0a51e6c0debc
```

Now, Alice and Bob have the same secret key (`Qabx` = `Qbax`). They can use symmetric-key cryptography for subsequent communications. A third party, Eve for example, knows $y^2 = x^3 + ax + b \bmod m$, P, Q_a, and Q_b, but cannot calculate the same secret key.

2.2. Point Addition and Point Doubling

Scalar point multiplication $Q = dP$ uses point addition (PA) and point doubling (PD).

2.2.1. Point Addition

Given $P = [x_p, y_p]$ and $Q = [x_q, y_q]$, the formulas for point addition $R = [x_r, y_r] = P + Q$ on elliptic curve $y^2 = x^3 + ax + b \bmod m$ are shown as follows, where λ is the slope of the line through points P and Q. The derivation of the formulas can be found in [18].

$$\begin{cases} \lambda = \dfrac{y_q - y_p}{x_q - x_p} \bmod m \\ x_r = (\lambda^2 - x_p - x_q) \bmod m \\ y_r = (\lambda(x_p - x_r) - y_p) \bmod m \end{cases} \tag{2}$$

The point at infinity, denoted $\mathcal{O}$, is included in the group of elliptic curves and is defined as $P + (-P) = \mathcal{O}$ for $Q = -P$. By this definition, $P + \mathcal{O} = P$. In our implementation,

$\mathcal{O}$ is represented as $[-1, -1]$. In the case of $P = \mathcal{O}$, $R = P + Q = \mathcal{O} + Q = Q$. In the case of $Q = \mathcal{O}$, $R = P + Q = P + \mathcal{O} = P$. We give the point addition $R = P + Q$ algorithm over the finite field of $\mathbb{F}_m$ in Algorithm 1. In the case of $Q = -P$, $R = P + Q = P + (-P) = \mathcal{O}$ (line 5 in the algorithm). In the case of $Q = P$, $R = P + Q = P + P = 2P$, we perform the point doubling $R = 2P$ (line 6 in the algorithm).

Algorithm 1 PA (P, Q, m, a) (Point Addition in Affine Coordinates).

inputs: Points $P = [P_x, P_y]$ and $Q = [Q_x, Q_y]$; m and a in $y^2 = x^3 + ax + b \bmod m$
output: $R = P + Q = [R_x, R_y] = [x_r, y_r]$
begin
1 $x_p = P_x, y_p = P_y, x_q = Q_x, y_q = Q_y, \mathcal{O} = [-1, -1]$
2 **if** $P = \mathcal{O}$ **return** Q /* $\mathcal{O} + Q = Q$ */
3 **if** $Q = \mathcal{O}$ **return** P /* $P + \mathcal{O} = P$ */
4 **if** $x_p = x_q$
5 **if** $(y_p + y_q) \bmod m = 0$ **return** $\mathcal{O}$ /* $P + (-P) = \mathcal{O}$ */
6 **else return** PD (P, m, a) /* $P + P = 2P$ */
7 $\lambda = ((y_q - y_p)/(x_q - x_p)) \bmod m$
8 $x_r = (\lambda^2 - x_p - x_q) \bmod m$
9 $y_r = (\lambda(x_p - x_r) - y_p) \bmod m$
10 **return** $[x_r, y_r]$ /* $R = P + Q$ */
end

An example of point addition $R = P + Q$ on the Secp256k1 curve is shown below, where $[P_x, P_y] = P$, $[Q_x, Q_y] = Q$, and $[R_x, R_y] = R$ in affine coordinates.

```
Px = 0xfc7dafb820a20da1a73c36465f2fe37bfd98ce4ef3a10a5df110abda03b20a3d
Py = 0xa442a2d1b8bde4a09e45725add5daae89e726b56f0e8fe6609dacaf5279b2564
Qx = 0xe106c069450b2663febb83e29b67fa93c4c48a45d5fbe7ce4ddb8ceb601fcc1d
Qy = 0xc9da9bd440909c8862c06a44d432d2dd45284636b7049b9bf4695f9e4018d2f2
Rx = 0xfd52a0334e16f8cf45a6b0820887a9e8b1b180516a76c8adfef95df98aeef376
Ry = 0xb0fe3f04cc4c64fd66a133b8c97b4905771238f8ba89631efb85a8059e969a49
```

2.2.2. Point Doubling

Given $P = [x_p, y_p]$, the formulas for point doubling $R = [x_r, y_r] = 2P$ on elliptic curve $y^2 = x^3 + ax + b \bmod m$ are shown as follows, where λ is the slope of the tangent line of the elliptic curve at point P. The derivation of the formulas can be found in [18].

$$\begin{cases} \lambda = \dfrac{3x_p^2 + a}{2y_p} \bmod m \\ x_r = (\lambda^2 - 2x_p) \bmod m \\ y_r = (\lambda(x_p - x_r) - y_p) \bmod m \end{cases} \tag{3}$$

We give the point doubling $R = 2P$ algorithm over the finite field of $\mathbb{F}_m$ in Algorithm 2. In the case of $P_y = 0$ (vertical tangent line), $R = 2P = \mathcal{O}$ (line 2 in the algorithm).

An example of point doubling $R = 2P$ on the Secp256k1 curve is shown below, where $[P_x, P_y] = P$ and $[R_x, R_y] = R$ in affine coordinates.

```
Px = 0x6034b56424fb31ea6ec5483b52ae5d07d6f3ef80264d769ae2714abb83fb279a
Py = 0xfe4cde1ff7546a87f906f50ab1002fda7811828ea6fc467a44d1c6c11aa65a37
Rx = 0x5491ee8b73a4ed9713ed32e467de5100b80861babf8ffd09fd595ab457d042c9
Ry = 0xf91e6a4e132a1bdf4f5c846559431ec7373de8872b719f188b5902932f0a2b30
```

The computation of λ in PA and PD requires modular division, which can be realized using a modular inversion algorithm based on the EEA.

Algorithm 2 PD (P, m, a) (Point Doubling in Affine Coordinates).

inputs: Point $P = [P_x, P_y]$; m and a in $y^2 = x^3 + ax + b \bmod m$
output: $R = 2P = [R_x, R_y] = [x_r, y_r]$
begin
1 $x_p = P_x, y_p = P_y, \mathcal{O} = [-1, -1]$
2 **if** $y_p = 0$ **return** $\mathcal{O}$ /* vertical tangent */
3 $\lambda = ((3x_p^2 + a)/(2y_p)) \bmod m$
4 $x_r = (\lambda^2 - 2x_p) \bmod m$
5 $y_r = (\lambda(x_p - x_r) - y_p) \bmod m$
6 **return** $[x_r, y_r]$ /* $R = 2P$ */
end

2.3. Modular Inversion Algorithms

Given a prime number m, the inverse r of a number a with $a < m$ is defined as

$$r = a^{-1} \bmod m \tag{4}$$

Algorithm 3 (modinv_fermat) implements the modular inversion calculation using Fermat's little theorem. If m is prime and $a \neq 0 \pmod m$, Fermat's little theorem says that $a^{m-1} = 1 \bmod m$. Multiplying both sides by a^{-1} gives us $a^{m-2} \bmod m = a^{-1} \bmod m$. Then, we can calculate the modular inversion with $r = a^{m-2} \bmod m$. This modular exponentiation can be performed using the multiply-squaring method, as shown in Algorithm 3 (modinv_fermat). This calculation consists of costly modular multiply and modular squaring, very similar to RSA exponentiation [19].

Algorithm 3 modinv_fermat (a, m) (Modinv using Fermat's little theorem).

inputs: Prime m and a with $a < m$
output: $a^{-1} \bmod m$
begin
1 $k = m - 2;\ x = 1;\ y = a$
2 **while** $k \neq 0$
3 **if** k is odd
4 $x = xy \bmod m$ /* modular multiply */
5 $y = y^2 \bmod m$ /* modular squaring */
6 $k = k \gg 1$
7 **return** x
end

The Python code below implements Algorithm 3 (modinv_fermat). When executed, it outputs 4 4 4. The first value is calculated by the code, and the rest are for checking.

```
# modinv_fermat.py, Fermat's Little Theorem, a^{-1} = a^{m-2} mod m
def modinv_fermat (a, m): # return a^{-1} mod m
    k = m - 2; x = 1; y = a
    while k != 0:
        if k & 1 == 1:
            x = x * y % m       # modular multiply
        y = y * y % m           # modular squaring
        k = k >> 1
    return x
a = 3; m = 11
print (modinv_fermat(a, m), pow(a, -1, m), pow(a, m-2, m))
"""
$ python3 modinv_fermat.py
4 4 4
"""
```

The EEA can be used for the modular inversion calculation. Algorithm 4 (modinv1) gives the fundamental EEA for the modular inversion calculation. Line 4 calculates the integer quotient q of u divided by v. Lines 5 and 6 calculate the remainders $v = u - qv$ and $y = x - qy$ based on the quotient q and store the original v and y in u and x, respectively. These calculations are repeated until $v = 0$.

Algorithm 4 modinv1 (b, a, m) (Modular Inversion Algorithm 1).

inputs: Prime m, a, and b with $a, b < m$
output: $ba^{-1} \bmod m$
begin
1 $u,\ v = a,\ m$ /* $u = a$ and $v = m$ */
2 $x,\ y = b,\ 0$ /* $x = b$ and $y = 0$ */
3 **while** $v \neq 0$
4 $q = \lfloor u/v \rfloor$ /* q: integer quotient */
5 $u,\ v = v,\ u - qv$
6 $x,\ y = y,\ x - qy$
7 **return** $x \bmod m$
end

Considering $b = 1$. u and x are initialized with a and 1, respectively. At each iteration, u and x are modified with similar calculations. Therefore, when u reaches 1 from a, x reaches the reciprocal of a from 1:

$$u : a \to (/a) \to 1; \qquad x : 1 \to (/a) \to a^{-1}; \qquad x : b \to (/a) \to ba^{-1}$$

If m is a prime number, the greatest common divisor of a and m is guaranteed to be 1, and we can always obtain the inverse result of a. With the initialization of x with b, the algorithm performs the modular division $r = ba^{-1} \bmod m$.

An execution example of Algorithm 4 (modinv1) with $b = 1$, $a = 3$, and $m = 11$ is shown in Table 2. The calculation finishes when $v = 0$. The result $r = a^{-1} \bmod m = x \bmod m = 4 \bmod 11 = 4$. We can check the correctness as follows: $ra \bmod m = 4 \times 3 \bmod 11 = 12 \bmod 11 = 1 \bmod 11$.

Table 2. Execution example of Algorithm 4 (modinv1) with $b = 1$, $a = 3$, and $m = 11$. It calculates $r = 3^{-1} \bmod 11$. The result is $x \bmod m = 4 \bmod 11 = 4$.

i	u	v	x	y	q
0	$3 = a$	$11 = m$	$1 = b$	0	$q = u/v$
0	$u = v$	$v = u - q * v$	$x = y$	$y = x - q * y$	
1					$0 = 3/11$
1	$11 = v$	$3 = 3 - 0 * 11$	$0 = y$	$1 = 1 - 0 * 0$	
2					$3 = 11/3$
2	$3 = v$	$2 = 11 - 3 * 3$	$1 = y$	$-3 = 0 - 3 * 1$	
3					$1 = 3/2$
3	$2 = v$	$1 = 3 - 1 * 2$	$-3 = y$	$4 = 1 - 1 * (-3)$	
4					$2 = 2/1$
4	$1 = v$	$0 = 2 - 2 * 1$	$4 = y$	$-11 = (-3) - 2 * 4$	
End	$u = 1$	$v = 0$	$x = 4$		

The algorithm requires division, which is expensive. As shown in Algorithm 5 (modinv2), we can eliminate the division by setting the quotient to 0 or 1.

Algorithm 5 modinv2 (b, a, m) (Modular Inversion Algorithm 2).

inputs: Prime m, a, and b with $a, b < m$
output: $ba^{-1} \bmod m$
begin
1 $u,\ v = a,\ m$
2 $x,\ y = b,\ 0$
3 **while** $v \neq 0$
4 $q = 0$ **if** $u < v$ **else** 1
5 $u,\ v = v,\ u - qv$
6 $x,\ y = y,\ x - qy$
7 **return** $x \bmod m$
end

The algorithm yields a quotient of 0 or 1 based on the comparison of u and v. If the quotient is a 1, subtractions $u - v$ and $x - y$ are performed (lines 5 and 6). Otherwise no calculations are performed (simply swapping u with v and swapping x with y), which make the computation slower. The execution of Algorithm 5 (modinv2) with $b = 1$, $a = 3$, and $m = 11$ requires nine iterations, as shown in Table 3.

Table 3. Execution example of Algorithm 5 (modinv2) with $b = 1$, $a = 3$, and $m = 11$. It calculates $r = 3^{-1} \bmod 11$. The result is $x \bmod m = -7 \bmod 11 = (11 - 7) \bmod 11 = 4$.

i	u	v	x	y	q
0	3	11	1	0	0
1	11	3	0	1	1
2	3	8	1	−1	0
3	8	3	−1	1	1
4	3	5	1	−2	0
5	5	3	−2	1	1
6	3	2	1	−3	1
7	2	1	−3	4	1
8	1	1	4	−7	1
9	1	0	−7	11	

The algorithm can be modified to remove the no calculations, as shown in Algorithm 6 (modinv3). The code in lines 4 and 5 ensures that u and v are non-negative. Note that u and x are one pair, and v and y are another pair. Algorithm 6 (modinv3) reduces the number of iterations by about half, as shown in Table 4.

Algorithm 6 modinv3 (b, a, m) (Modular Inversion Algorithm 3).

inputs: Prime m, a, and b with $a, b < m$
output: $ba^{-1} \bmod m$
begin
1 $u,\ v = a,\ m$
2 $x,\ y = b,\ 0$
3 **while** $u \neq 1$ and $v \neq 1$
4 **if** $u < v:\ v,\ y = v - u,\ y - x$
5 **else** $u,\ x = u - v,\ x - y$
6 **if** $u = 1$ **return** $x \bmod m$
7 **else** **return** $y \bmod m$
end

Table 4. Execution example of Algorithm 6 (modinv3) with $b = 1$, $a = 3$, and $m = 11$. It calculates $r = 3^{-1} \bmod 11$. The result is $x \bmod m = 4 \bmod 11 = 4$, because $u = 1$.

i	u	v	x	y
0	3	11	1	0
1	3	8	1	−1
2	3	5	1	−2
3	3	2	1	−3
4	1	2	4	−3

In Table 4, because $v > u$ for $i = 0$ to 2, we update v with $v - u$. Correspondingly, we update y with $y - x$. For $i = 3$, because $u > v$, we update u with $u - v$. Correspondingly, we update x with $x - y$. For $i = 4$, because $u = 1$, the result is $x \bmod m = 4$.

We can check u first, before the subtractions. If it is even, we shift it to the right by one bit (the least significant bit 0 is shifted out). Correspondingly, x must also be shifted. To ensure that the value to be right-shifted is even, m will be added before the shift if x is odd. Note that m is odd because it is a prime number. These shifts of u and x can be performed repeatedly until u becomes an odd number.

Similarly, if v is even, we shift it to the right by one bit. Correspondingly, y must also be shifted. If y is odd, m will be added before the shift. These shifts of v and y can be performed repeatedly until v becomes an odd number.

Then, we obtain the algorithm shown in Algorithm 7 (modinv4). The idea behind it is that division makes u and v reach 1 faster than subtraction. In fact, this is Algorithm 2.22 provided in [2] and implemented in Verilog HDL in [18].

Algorithm 7 modinv4 (b, a, m) (Modular Inversion Algorithm 4).

inputs: Prime m, a, and b with $a, b < m$
output: $ba^{-1} \bmod m$
begin

```
 1    u, v = a, m
 2    x, y = b, 0
 3    while u ≠ 1 and v ≠ 1
 4        while u is even
 5            u = u/2
 6            if  x is even: x = x/2
 7            else           x = (x + m)/2
 8        while v is even
 9            v = v/2
10            if  y is even: y = y/2
11            else           y = (y + m)/2
12        if  u < v :  v, y = v − u, y − x
13        else         u, x = u − v, x − y
14    if  u = 1 return x mod m
15    else      return y mod m
```

end

Algorithm 7 (modinv4) has been widely adopted in ECC implementations. When the two inner while loops (lines 4 to 11) finish, u and v are both odd numbers. Therefore, $u - v$ or $v - u$ is even. Then, we can shift it to the right by one bit. Correspondingly, $x - y$ or $y - x$ must also be shifted. If $x - y$ or $y - x$ is odd, m must be added before the shift so that the bit being shifted out is 0. This algorithm is shown in Algorithm 8 (modinv5). The shifts are performed by the code in lines 12 to 19.

Algorithm 8 modinv5 (b, a, m) (Modular Inversion Algorithm 5).

inputs: Prime m, a, and b with $a, b < m$
output: $ba^{-1} \bmod m$
begin
```
 1    u, v = a, m
 2    x, y = b, 0
 3    while u ≠ 1 and v ≠ 1
 4        while u is even
 5            u = u/2
 6            if x is even: x = x/2
 7            else          x = (x + m)/2
 8        while v is even
 9            v = v/2
10            if y is even: y = y/2
11            else          y = (y + m)/2
12        if u < v
13            v, y = (v − u)/2, y − x
14            if y is even: y = y/2
15            else          y = (y + m)/2
16        else
17            u, x = (u − v)/2, x − y
18            if x is even: x = x/2
19            else          x = (x + m)/2
20    if u = 1 return x mod m
21    else     return y mod m
```
end

The two inner while loops in Algorithm 8 (modinv5) can be replaced by assigning u, x, v, y, or 0 to the temporary variables tu, tx, tv, ty, so that $tu - tv$ is an even number. Next, $tu - tv$ is shifted one bit to the right. Correspondingly, $tx - ty$ is also shifted one bit to the right. Note that if $tx - ty$ is odd, m is added before the shift. This algorithm is shown in Algorithm 9 (modinv6). Note that only $tx - ty$ is executed; it may be negative. The code in lines 12 and 13 ensures that u and v are non-negative.

Algorithm 9 modinv6 (b, a, m) (Modular Inversion Algorithm 6).

inputs: Prime m, a, and b with $a, b < m$
output: $ba^{-1} \bmod m$
begin
```
 1    u, v = a, m
 2    x, y = b, 0
 3    while u ≠ 1 and v ≠ 1
 4        if u is odd: tv, ty = v, y
 5        else         tv, ty = 0, 0
 6        if v is odd: tu, tx = u, x
 7        else         tu, tx = 0, 0
 8        tuv, txy = tu − tv, tx − ty
 9        uv = tuv/2
10        if txy is even: xy = txy/2
11        else            xy = (txy + m)/2
12        if uv < 0 :  v, y = −uv, −xy
13        else         u, x = uv, xy
14    if u = 1 return x mod m
15    else     return y mod m
```
end

Algorithm 9 (modinv6) requires calculations of $tu - tv$, $tv - tu$, $tx - ty$, and $ty - tx$. We can unify these calculations with negative assignments $-v$ and $-y$ to v and y, respectively, so that only $tu + tv$ and $tx + ty$ are sufficient for the calculations. That is, with negative assignments $-v$ and $-y$ to v and y, $u = u - v = u + (-v)$ becomes $u = u + v$, and $x = x - y = x + (-y)$ becomes $x = x + y$. Similarly, $v = -(u - v)$ becomes $v = u + v$, and $y = -(x - y)$ becomes $y = x + y$ with negative assignments $-v$ and $-y$ to v and y. Therefore, $u + v$ and $x + y$ are sufficient for the calculations, saving hardware costs.

The algorithm is given in Algorithm 10 (modinv7). Because of the negative assignments to v and y, v is initialized with $-m$ and y is initialized with $-0 = 0$. Note that x is never greater than or equal to m. Therefore, for the final result, no adjustment of $x = x - m$ or $x = x \bmod m$ is required. All we need is $x = x + m$ for $x < 0$.

Algorithm 10 modinv7 (b, a, m) (Modular Inversion Algorithm 7).

inputs: Prime m, a, and b with $a, b < m$
output: $ba^{-1} \bmod m$
begin

```
 1    u, v = a, −m
 2    x, y = b, −0
 3    while u ≠ 1
 4        if u is odd: tv, ty = v, y
 5        else         tv, ty = 0, 0
 6        if v is odd: tu, tx = u, x
 7        else         tu, tx = 0, 0
 8        tuv, txy = tu + tv, tx + ty
 9        uv = tuv/2
10        if txy is even: xy = txy/2
11        else
12            if txy < 0 : xy = (txy + m)/2
13            else         xy = (txy − m)/2
14        if uv < 0 : v, y = uv, xy
15        else        u, x = uv, xy
16    if x < 0 : x = x + m
17    return x
end
```

The Python codes for Algorithms 4–10 (modinv1 to modinv7) are given in Appendix A. We generate random numbers b and a that are smaller than a fixed 256-bit m. For the same b, a, m inputs, all modular inversion algorithms have the same output.

A modular inversion algorithm is said to be good when u reaches 1 quickly (high performance) and the algorithm uses a small number of adders and subtractors (low cost).

3. Proposed Radix-8 Modular Inversion Algorithm and Its Performance

The proposed mixed radix-8 modular inversion algorithm is given in Algorithm 11 (modinv_radix8). To calculate $r = ba^{-1} \bmod m$, we initialize $u = a$, $v = -m$, $x = b$, and $y = -0$ with the negative assignment to v and y. The temporary variable tu is assigned with u or 0 and the temporary variable tv is assigned with v or 0, so that $tuv = tu + tv$ is even. Correspondingly, the temporary variable tx is also assigned with x or 0 and the temporary variable ty is assigned with y or 0. If the least significant three bits of tuv are 000, it is shifted to the right by three bits (radix-8). Otherwise, if the least significant two bits of tuv are 00, it is shifted to the right by two bits (radix-4). Otherwise, it is shifted to the right by one bit (radix-2), because tuv is even. This is also called a hybrid radix algorithm. It is difficult to develop a complete radix-8 algorithm without using radix-4 or radix-2 arithmetic. We need to handle all cases where the least significant three bits of tuv are not 000 (there are seven cases) and perform the corresponding radix-8 arithmetic.

Algorithm 11 modinv_radix8 (b, a, m) (Radix-8 Modular Inversion Algorithm).

inputs: Prime m, a, and b with $a, b < m$
output: $ba^{-1} \bmod m$
begin

```
 1    u, v = a, −m
 2    x, y = b, −0
 3    while u ≠ 1
 4        if u is odd: tv, ty = v, y
 5        else         tv, ty = 0, 0
 6        if v is odd: tu, tx = u, x
 7        else         tu, tx = 0, 0
 8        tuv, txy = tu + tv, tx + ty                          /* tuv is even */
 9        if tuv & 6 = 0                                       /* radix 8 */
10            uv = tuv/8
11            if txy & 1 = 0
12                if txy & 2 = 0
13                    if txy & 4 = 0 :  xy = txy/8
14                    else              xy = (txy + 4m)/8
15                else
16                    if txy & 4 = (2m & 4) :  xy = (txy − 2m)/8
17                    else                     xy = (txy + 2m)/8
18            else
19                if txy & 6 = m & 6 :  xy = (txy − m)/8
20                else
21                    if txy & 2 = m & 2 :  xy = (txy + 3m)/8
22                    else
23                        if txy & 4 ≠ m & 4 :  xy = (txy + m)/8
24                        else                  xy = (txy − 3m)/8
25        else
26            if tuv & 2 = 0                                   /* radix 4 */
27                uv = tuv/4
28                if txy & 1 = 0
29                    if txy & 2 = 0 :  xy = txy/4
30                    else              xy = (txy + 2m)/4
31                else
32                    if txy & 3 = m & 3 :  xy = (txy − m)/4
33                    else                  xy = (txy + m)/4
34            else                                             /* radix 2 */
35                uv = tuv/2
36                if txy & 1 = 0 :  xy = txy/2
37                else
38                    if txy < 0 :  xy = (txy + m)/2
39                    else          xy = (txy − m)/2
40        if  uv < 0 :  v, y = uv, xy
41        else          u, x = uv, xy
42    if  x < 0 :  x = x + m
43    return x
end
```

Correspondingly, tx and ty are arranged and $txy = tx + ty$ is also shifted to the right by three bits, two bits, or one bit. The bits being shifted out must be 0. Therefore, we need to adjust txy using the prime number m before the shift. Table 5 lists such adjustments based on the least significant three bits of txy and the least significant three bits of m for the radix-8 operations, where x represents a don't-care term. The least significant three bits of the adjusted value are 000, as shown in the Comment column of the table.

Table 5. XY adjustment for shift right by three bits in the proposed modular inversion algorithm.

txy	m	$2m$	$3m$	$4m$	xy	Comment
000	xx1				txy / 8	$0+0=0$
100	xx1			100	$(txy+4m)$ / 8	$4+4=8$
010	x01	010			$(txy-2m)$ / 8	$2-2=0$
110	x11	110			$(txy-2m)$ / 8	$6-6=0$
010	x11	110			$(txy+2m)$ / 8	$2+6=8$
110	x01	010			$(txy+2m)$ / 8	$6+2=8$
001	001				$(txy-m)$ / 8	$1-1=0$
011	011				$(txy-m)$ / 8	$3-3=0$
101	101				$(txy-m)$ / 8	$5-5=0$
111	111				$(txy-m)$ / 8	$7-7=0$
001	101	010	111		$(txy+3m)$ / 8	$1+7=8$
011	111	110	101		$(txy+3m)$ / 8	$3+5=8$
101	001	010	011		$(txy+3m)$ / 8	$5+3=8$
111	011	110	001		$(txy+3m)$ / 8	$7+1=8$
001	111				$(txy+m)$ / 8	$1+7=8$
011	101				$(txy+m)$ / 8	$3+5=8$
101	011				$(txy+m)$ / 8	$5+3=8$
111	001				$(txy+m)$ / 8	$7+1=8$
001	011	110	001		$(txy-3m)$ / 8	$1-1=0$
011	001	010	011		$(txy-3m)$ / 8	$3-3=0$
101	111	110	101		$(txy-3m)$ / 8	$5-5=0$
111	101	010	111		$(txy-3m)$ / 8	$7-7=0$

Similarly, Table 6 lists the adjustments based on the least significant two bits of txy and the least significant two bits of m for the radix-4 operations. The least significant two bits of the adjusted value are 00, as shown in the Comment column of the table.

Table 6. XY adjustment for shift right by two bits in the proposed modular inversion algorithm.

txy	m	$2m$	xy	Comment
00	x1		txy / 4	$0+0=0$
10	x1	10	$(txy+2m)$ / 4	$2+2=4$
01	01		$(txy-m)$ / 4	$1-1=0$
11	11		$(txy-m)$ / 4	$3-3=0$
01	11		$(txy+m)$ / 4	$1+3=4$
11	01		$(txy+m)$ / 4	$3+1=4$

The Python code for the proposed algorithm is also given in Appendix A. The modinv_radix8 algorithm takes 206 iterations to reach $u = 1$ and $v = -1$. In contrast, the modinv_radix4 and modinv_radix2 algorithms require 243 and 356 iterations, respectively.

To reduce the number of adders, we use a multiplexer to select an appropriate value and assign it to the temporary variable tz. Then, we perform $txy = tx + ty + tz$. Based on the least significant three bits of tuv, we assign $txy \gg 1$, $txy \gg 2$, or $txy \gg 3$ to xy.

Figure 1 shows the block diagram of the proposed radix-8 modular inversion circuit. To perform the addition $txy = tx + ty + tz$, $-2m$ and $-3m$ are replaced by $+6m$ and $+5m$, respectively. This is because, for example, for an integer i, $(i - 3m) \bmod m = (i - 3m + 8m) \bmod m = (i + 5m) \bmod m$. Furthermore, for the addition, we prepare $-m$ that can be obtained by inverting all the bits of m and setting the right-most bit to 1 because m is odd. The Verilog HDL implementation uses continuous assignment to compute uv and xy and writes them to the corresponding registers on the rising edge of the clock signal. Note that we have to use adders for generating $3m$, $5m$, and $6m$, which are not shown in the figure.

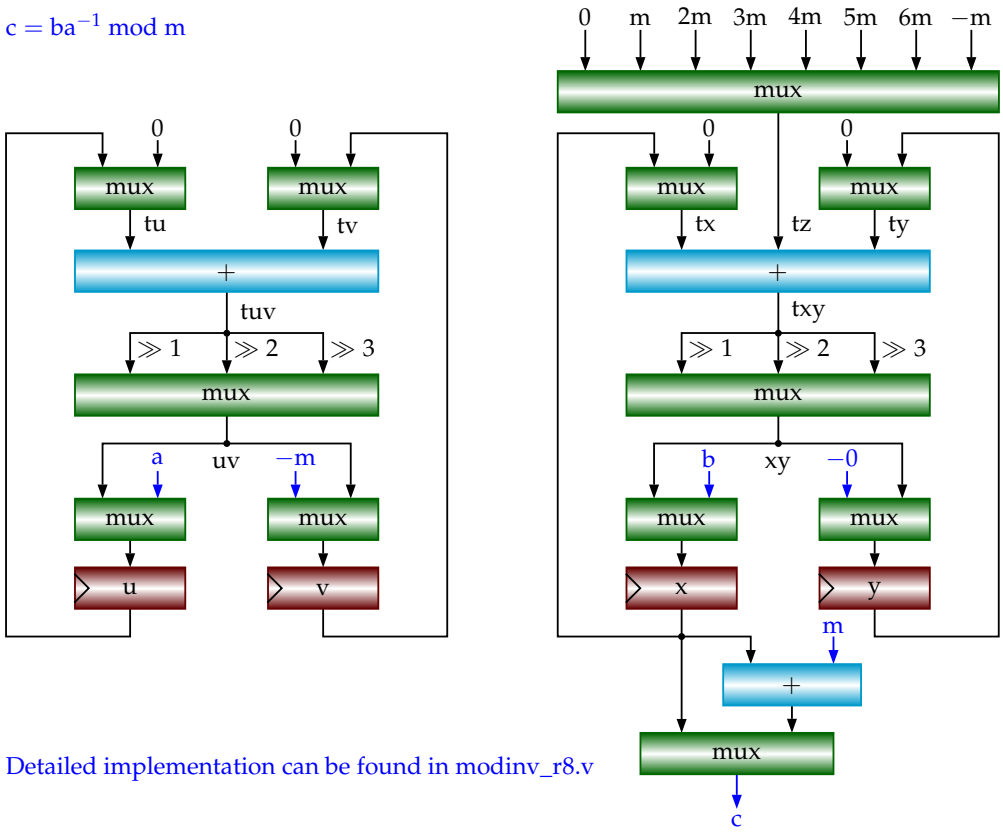

Figure 1. Block diagram of the proposed mixed radix-8 modular inversion circuit.

Below we give the hardware implementation code in Verilog HDL for the proposed radix-8 modular inversion algorithm (modinv_r8.v). The signals `start` and `ready` indicate the start of the modular inversion calculation and the availability of the calculation result, respectively. Because we use the Secp256k1 elliptic curve, the input and output signals b, a, m, and c are 256 bits. We use 260 bits for the internal signals. Instead of the 260-bit $txy = tx + ty$ in the Python code, we achieve it with 3 bits (`t3`). This reduces the execution time and hardware costs. And we perform $(tx < 0) \,|\, (ty < 0)$ for $txy < 0$. This guarantees $(tx + ty + m)/2 < m$.

```
`timescale 1ns/1ns // proposed radix-8 implementation for c = b * a^{-1} mod m
module modinv_r8 (clk, rst_n, start, b, a, m, c, ready, busy, ready0);
    input          clk, rst_n;
    input          start;
    input  [255:0] b, a, m;
    output [255:0] c;
    output         ready, ready0;
    output reg     busy;
    reg            ready0,  ready1;
    assign ready = ready0 ^ ready1;
    reg    [259:0] u, v, x, y;                                  // registers
    wire   [259:0] p   = {4'h0,m};                              // p = m
    wire   [259:0] mm  = {4'hf,~m[255:1],1'b1};                 // mm = -m
    wire   [259:0] tu  = v[0] ? u : 0;
    wire   [259:0] tx  = v[0] ? x : 0;
    wire   [259:0] tv  = u[0] ? v : 0;
    wire   [259:0] ty  = u[0] ? y : 0;
    wire   [259:0] tuv = tu + tv;                               // adder for uv
    wire   [259:0] uv2 = {tuv[259],tuv[259:1]};                 // tuv // 2
    wire   [259:0] uv4 = {{2{tuv[259]}},tuv[259:2]};            // tuv // 4
    wire   [259:0] uv8 = {{3{tuv[259]}},tuv[259:3]};            // tuv // 8
    wire   [259:0] uv  = tuv[1] ? uv2 : tuv[2] ? uv4 : uv8;     // uv
    wire     [2:0] t3  = tx[2:0] + ty[2:0];                     // t3 & 7
```

```
    wire           equ = t3[1:0] == p[1:0];                    // t3 & 3 == m & 3
    wire   [259:0] m2  = {p[258:0],1'b0};                      // 2m
    wire   [259:0] m4  = {p[257:0],2'b0};                      // 4m
    wire   [259:0] m3  = m2 + p;                               // 3m adder
    wire   [259:0] m5  = m4 + p;                               // 5m adder
    wire   [259:0] m6  = m4 + m2;                              // 6m adder
    wire   [259:0] tz2 = t3[0] ? (tx[259]|ty[259]) ? p : mm : 260'h0; // z2
    wire   [259:0] tz4 = t3[0] ? equ ? mm : p : t3[1] ? m2 : 260'h0;  // z4
    wire   [259:0] tz8 = t3[0] ? t3[2:1] == p[2:1] ?                  // z8
                         mm : t3[1] == p[1] ? m3 : t3[2] != p[2] ?
                         m  : m5 : t3[1] ? t3[2] == p[1] ?
                         m6 : m2 : t3[2] ? m4 : 260'h0;
    wire   [259:0] tz  = tuv[1] ? tz2 : tuv[2] ? tz4 : tz8;    // tz
    wire   [259:0] txy = tx + ty + tz;                         // adder for xy
    wire   [259:0] txy2= {txy[259],txy[259:1]};                // txy // 2
    wire   [259:0] txy4= {{2{txy[259]}},txy[259:2]};           // txy // 4
    wire   [259:0] txy8= {{3{txy[259]}},txy[259:3]};           // txy // 8
    wire   [259:0] xy  = tuv[1] ? txy2 : tuv[2] ? txy4 : txy8; // xy
    wire   [259:0] xpp = x + p;                                // x + m
    wire   [259:0] r   = x[259] ? xpp : x;                     // x + m ? x ?
    assign         c   = r[255:0];                             // result c
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin                                      // reset
            ready0 <= 0;
            ready1 <= 0;
            busy   <= 0;
        end else begin
            ready1 <= ready0;
            if (start) begin                                   // load
                u <= {4'b0,a};                                 // u <=  a
                v <= mm;                                       // v <= -m
                x <= {4'b0,b};                                 // x <=  b
                y <= {260'b0};                                 // y <=  0
                ready0 <= 0;
                ready1 <= 0;
                busy   <= 1;
            end else begin
                if (u == 1) begin                              // if u == 1
                    ready0 <= 1;                               //     ready0 = 1
                    busy   <= 0;                               //     busy   = 0
                end else begin                                 // else
                    if (uv[259]) begin                         //     if uv < 0
                        v <= uv;                               //         v = uv
                        y <= xy;                               //         y = xy
                    end else begin                             //     else
                        u <= uv;                               //         u = uv
                        x <= xy;                               //         x = xy
                    end
                end
            end
        end
    end
endmodule
```

Below is the testbench Verilog HDL code used to simulate modinv_r8.v.

```
`timescale 1ns/1ns
module modinv_r8_tb;
    reg          clk, rst_n, start;
    reg  [255:0] b, a, m;
    wire [255:0] c;
    wire         ready, busy, ready0;
    modinv_r8 inst (clk, rst_n, start, b, a, m, c, ready, busy, ready0);
    initial begin
        clk   = 1;
        rst_n = 0;
```

```
        start = 0;
        b = 256'h9cfa1c993911914be0f15bd74a878abe0079c6254b961b82e1abda76387d1d85;
        a = 256'hd5076ae274e874c2eb0f7778717c39460236549ddd9fc651e68a0c0e787b4ce8;
        m = 256'hffffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f;
        #1 rst_n = 1;
        #0 start = 1;
        #2 start = 0;
        wait(ready); // 416ns
        #40 $stop;
    end
    always #1 clk = !clk;
endmodule
```

Figure 2 shows the functional simulation waveform, generated with ModelSim. The result c was available in 416 ns. That is, the calculation took 208 clock cycles. Note that the value of the result c is the same as the output of the Python code in Appendix A.

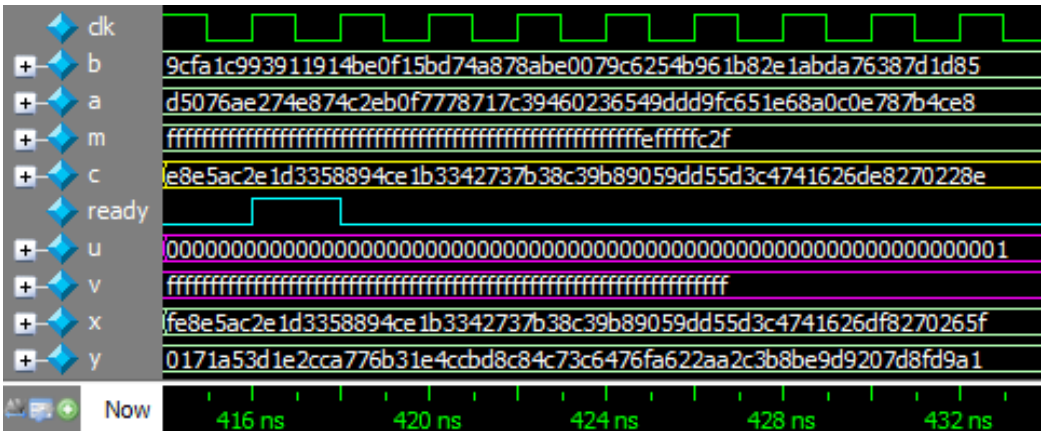

Figure 2. Waveform of the proposed radix-8 modular inversion algorithm.

We implemented the modular inversion algorithms on the Altera Cyclone V 5CGXFC9 E7F35C8 FPGA chip. The EDA tool we used is Quartus Prime Version 20.1.1 Build 720 11/11/2020 SJ Lite Edition. This is the latest edition that integrates with ModelSim for simulation. All algorithms were evaluated in the same environment.

Table 7 lists the cost performance of the modular inversion algorithms. The column Cycles shows the required number of clock cycles when executing the modular inversion algorithm. The column Freq.(MHz) shows the clock frequency in MHz at which the circuit can work. The column Latency(µs) shows the execution time in microseconds calculated by dividing the clock cycles by the clock frequency. The column ALMs shows the required number of adaptive logic modules. The column Registers shows the required number of flip-flops. The flip-flops are mainly used to store u, v, x, and y. Their contents are updated in every clock cycle. The last column shows the AT factor, which is the product of the Latency in milliseconds and the sum of the ALMs and Registers:

$$\text{AT} = \text{Latency} \times (\text{ALMs} + \text{Registers}) \tag{5}$$

The row [1] in the table shows the performance and cost of modular inversion using Fermat's little theorem $r = a^{m-2} \bmod m$. It consists of costly modular multiply and modular squaring. Its AT factor is much higher than the others. The remaining rows show the performance and cost of the EEA-based modular inversion algorithms. The numbers of registers used by [2,3,8,10] were larger than the others. This is because extra registers are used to adjust the value of x or y, so that the modular inversion result is within the range of 0 and m. Our algorithm implementation achieved an execution time of 3.67 µs and an AT factor of 8.30, outperforming all other implementations. Figure 3 shows an intuitive view of the latency and AT histograms.

Table 7. Comparison of modular inversion algorithms (on Altera Cyclone V FPGA chip).

Algorithm	Cycles	Freq. (MHz)	Latency (µs)	ALMs	Registers	AT
[1] 2011, Burton	66,264	57.54	1151.63	2004	2775	5503.66
[2] 2004, Hankerson	534	54.66	9.77	2619	1302	38.31
[3] 2015, Hossain	535	54.52	9.81	3735	1303	49.42
[4] 2005, Daly	358	39.73	9.01	2474	1038	31.64
[5] 2017, Mrabet	1205	64.55	18.67	1596	1043	49.26
[6] 2009, Chen	723	72.21	10.01	1968	1042	30.13
[7] 2017, Choi	358	63.60	5.63	959	1037	11.24
[8] 2023, Wang	423	59.56	7.10	3475	1303	33.92
[9] 2019, Yang	356	60.43	5.89	3950	1057	29.50
[10] 2007, Yan	423	54.99	7.69	3644	1303	38.05
[11] 2018, Dong	334	56.93	5.87	5276	1057	37.15
[12] 2022, Hao	534	54.66	9.77	2619	1302	38.31
[13] 2023, Guo	356	33.95	10.49	1653	1039	28.23
Ours	208	56.71	3.67	1227	1037	8.30

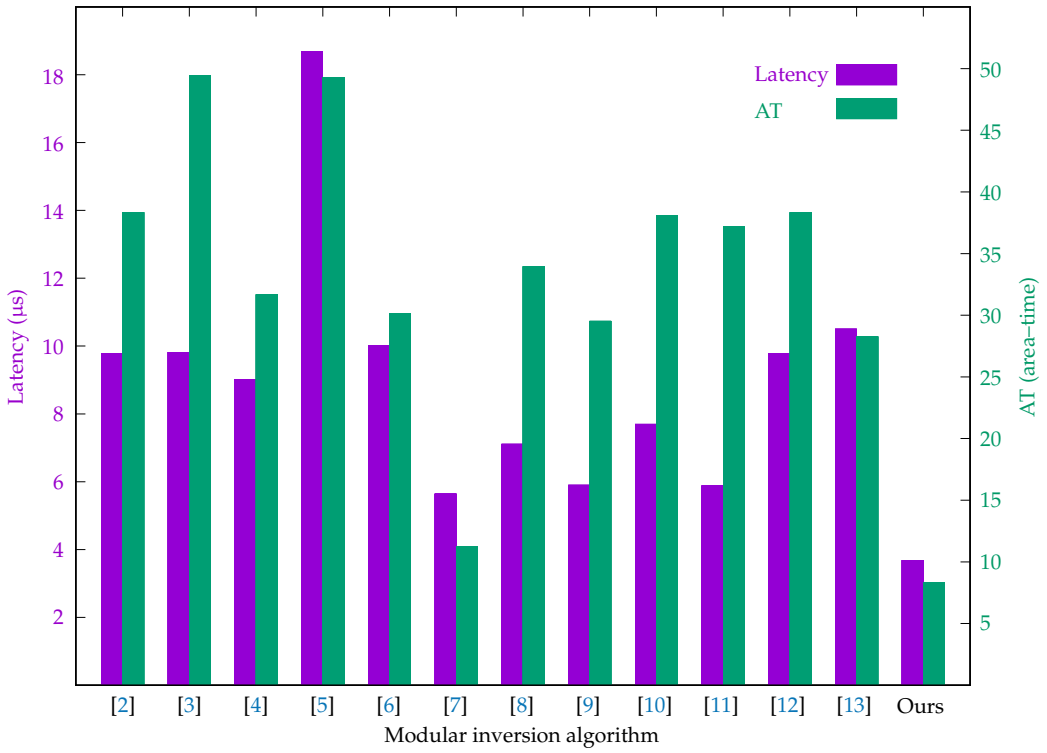

Figure 3. Latency and AT comparison of modular inversion algorithms. Details (year and first author's name) of the numbers [*n*] (algorithm) on the horizontal axis are in Table 7.

The proposed radix-8 algorithm was demonstrated to be efficient for 256-bit primes. Table 8 compares the cost performance on the Secp192k1 192-bit prime field curve and Secp521r1 521-bit prime field curve [16]. Here, we only provide a comparison with Algorithm 2.22 [2]. For ease of comparison, we also show the case when using the Secp256k1 256-bit prime field curve in the table. Our algorithm's AT outperformed [2] by $17.57/4.38 = 4.01$ times, $38.31/8.30 = 4.62$ times, and $253.31/55.44 = 4.57$ times for the curves of Secp192k1, Secp256k1, and Secp521r1, respectively. This shows that our

algorithm also demonstrated scalability to other prime sizes and adaptability to other cryptographic curves.

Table 8. Comparison on Secp192k1 192-bit and Secp521r1 521-bit prime field curves.

Curve	Algorithm	Cycles	Freq. (MHz)	Latency (µs)	ALMs	Registers	AT
Secp192k1	[2]	404	67.78	5.96	1965	982	17.57
	Ours	151	59.38	2.54	940	781	4.38
Secp256k1	[2]	534	54.66	9.77	2619	1302	38.31
	Ours	208	56.71	3.67	1227	1037	8.30
Secp521r1	[2]	1109	36.36	30.50	5678	2627	253.31
	Ours	429	33.60	12.77	2245	2097	55.44

4. ECC Implementation with Proposed Modular Inversion Algorithm

ECC relies on scalar point multiplication. Suppose $P = [x_p, y_p]$ is a point on the curve, the scalar point multiplication $Q = dP$ obtains the $Q = [x_q, y_q]$ that is also on the curve, where $d = \langle d_{n-1} \cdots d_1 d_0 \rangle$ is an n-bit scalar. Scalar point multiplication can be conducted with PA and PD, as shown in Algorithm 12.

Algorithm 12 ScaMul (d, P, m, a) (Scalar Point Multiplication).

inputs: $d = \langle d_{n-1} \cdots d_1 d_0 \rangle$ and point $P = [P_x, P_y]$; m and a in $y^2 = x^3 + ax + b \bmod m$
output: $Q = dP$
begin
1 $Q = \mathcal{O}, R = P, k = d$ /* $Q = \mathcal{O}$ and $R = P$ */
2 **while** $k \neq 0$ **do**
3 **if** $k_0 = 1$
4 $Q =$ PA (Q, R, m, a) /* $Q = Q + R$ (Algorithm 1) */
5 $R =$ PD (R, m, a) /* $R = 2R$ (Algorithm 2) */
6 $k = k \gg 1$
7 **endwhile**
8 **return** Q /* $Q = dP$ */
end

The algorithm calls point addition PA (P, Q, m, a) and point doubling PD (P, m, a). Table 9 gives an example to show the calculation steps of the scalar point multiplication. For a 5-bit $d = 10101_2 = 21$, we calculate $Q = dP$ in five steps to obtain $Q = 21P$. We can see that the algorithm is similar to RSA exponentiation [19].

Table 9. Execution example of $Q = dP$ with $d = 10101_2 = 21$.

	Weight	Point Addition	Point Doubling
Initial		$Q = \mathcal{O}$	$R = P$
$d_0 = 1$	1	$Q = Q + R = \mathcal{O} + P = P$	$R = 2R = 2P$
$d_1 = 0$	2		$R = 2R = 4P$
$d_2 = 1$	4	$Q = Q + R = P + 4P = 5P$	$R = 2R = 8P$
$d_3 = 0$	8		$R = 2R = 16P$
$d_4 = 1$	16	$Q = Q + R = 5P + 16P = 21P$	$R = 2R = 32P$

We used the ECDH key exchange protocol, described in Table 1, to establish a shared secret key for two parties. An ECDH key exchange algorithm in Python code is given in Appendix B. The algorithm invokes the scalar point multiplications that use two computations — point addition and point doubling. Four primitive modular calculations (addition,

subtraction, multiplication, and inversion) are used for these two computations, as shown in Figure 4. The Python function names in Appendix B are also shown in the figure.

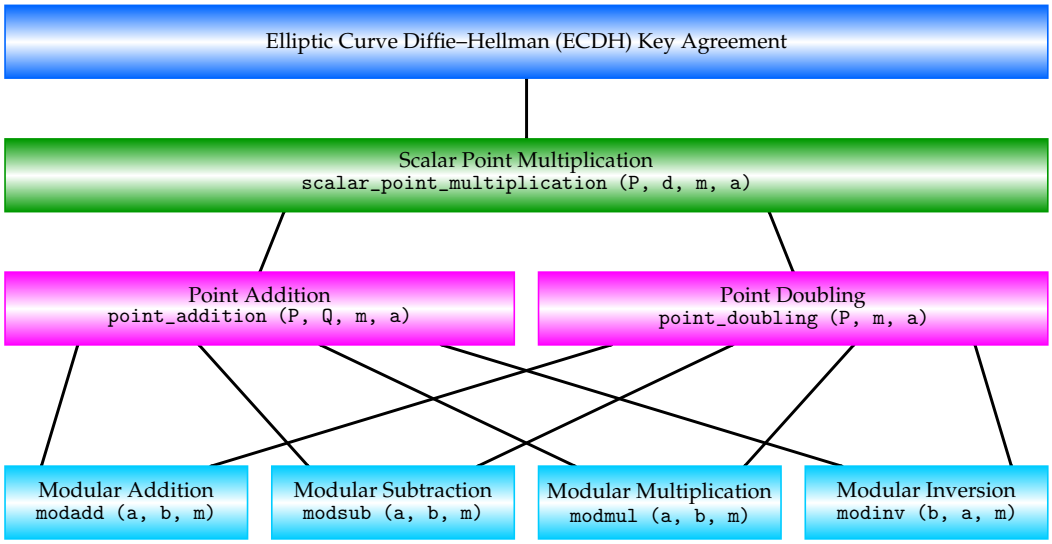

Figure 4. The ECDH key exchange algorithm uses scalar point multiplication, which uses two operations, point addition and point doubling, which use the four basic modular operations (addition, subtraction, multiplication, and inversion).

See Appendix B, the Python code is hardware-oriented. Essentially, a Python function defined using the `def` keyword was implemented in a Verilog HDL module. For integrity, we listed the modinv_radix8 code again, but now with $+6m$ and $+5m$ instead of $-2m$ and $-3m$, respectively, and the function name has been changed to modinv.

Based on the Python code, we implemented ECC using our radix-8 modular inversion algorithm for calculating λ in PA and PD. Figure 5 shows the functional simulation waveform of scalar point multiplication $Q = dP$ with $P = [x, y]$ and $Q = [qx, qy]$. The result was available in 635,362 ns. That is, the calculation took 317,681 clock cycles. Outputs qx and qy are the same as the outputs `Qax` and `Qay`, respectively, of the Python code in Appendix B.

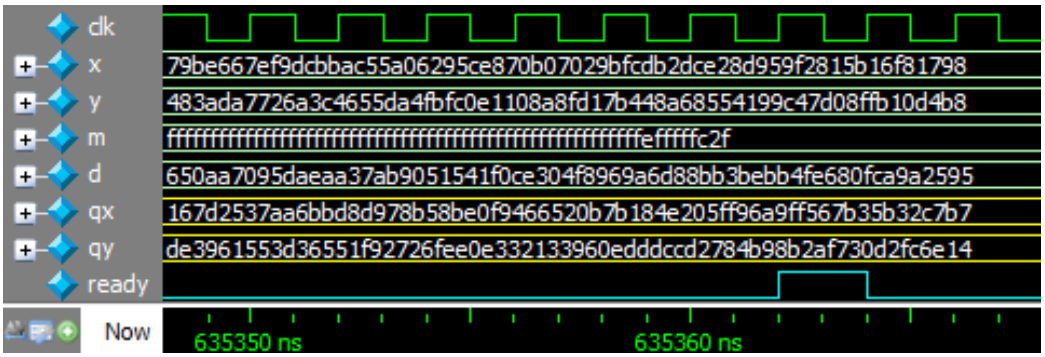

Figure 5. Waveform of the scalar point multiplication $Q = dP$ with $P = [x, y]$ and $Q = [qx, qy]$ using proposed the radix-8 modular inversion algorithm.

An ECC cost performance comparison is given in Table 10 when implementing on the Altera Cyclone V 5CGXFC9E7F35C8 FPGA chip. We also used Quartus Prime Version 20.1.1 Lite Edition for the implementation. All ECC implementations used the same circuit, except for the modular inversion part.

Figure 6 shows the latency and AT histogram. The ECC latency using our proposed radix-8 modular inversion algorithm was 0.01970 s and its AT factor was 393,546.29. From the table and histogram, we can see that our ECC implementation achieved lower latency and lower AT factor than all other implementations.

Table 10. ECC comparison using modular inversion algorithms (on Altera Cyclone V FPGA chip).

Algorithm	Cycles	Freq. (MHz)	Latency (ms)	ALMs	Registers	AT
[2] 2004, Hankerson	402,145	15.94	25.23	15,043	8355	590,300.42
[3] 2015, Hossain	402,400	15.58	25.83	17,585	8355	669,977.92
[4] 2005, Daly	357,262	16.06	22.25	14,975	7821	507,107.38
[5] 2017, Mrabet	570,142	16.07	35.48	13,292	7834	749,522.08
[6] 2009, Chen	455,425	16.15	28.20	14,211	7831	621,577.58
[7] 2017, Choi	356,878	16.01	22.29	12,114	7820	444,347.67
[8] 2023, Wang	372,127	15.98	23.29	17,292	8353	597,196.30
[9] 2019, Yang	352,761	15.72	22.44	17,841	7860	576,737.31
[10] 2007, Yan	372,127	16.08	23.14	18,548	8355	622,595.32
[11] 2018, Dong	346,194	15.88	21.80	20,716	7859	622,952.99
[12] 2022, Hao	402,145	15.89	25.31	15,041	8354	592,081.96
[13] 2023, Guo	356,496	15.64	22.79	13,571	7824	487,674.68
Ours	317,681	16.13	19.70	12,160	7822	393,546.29

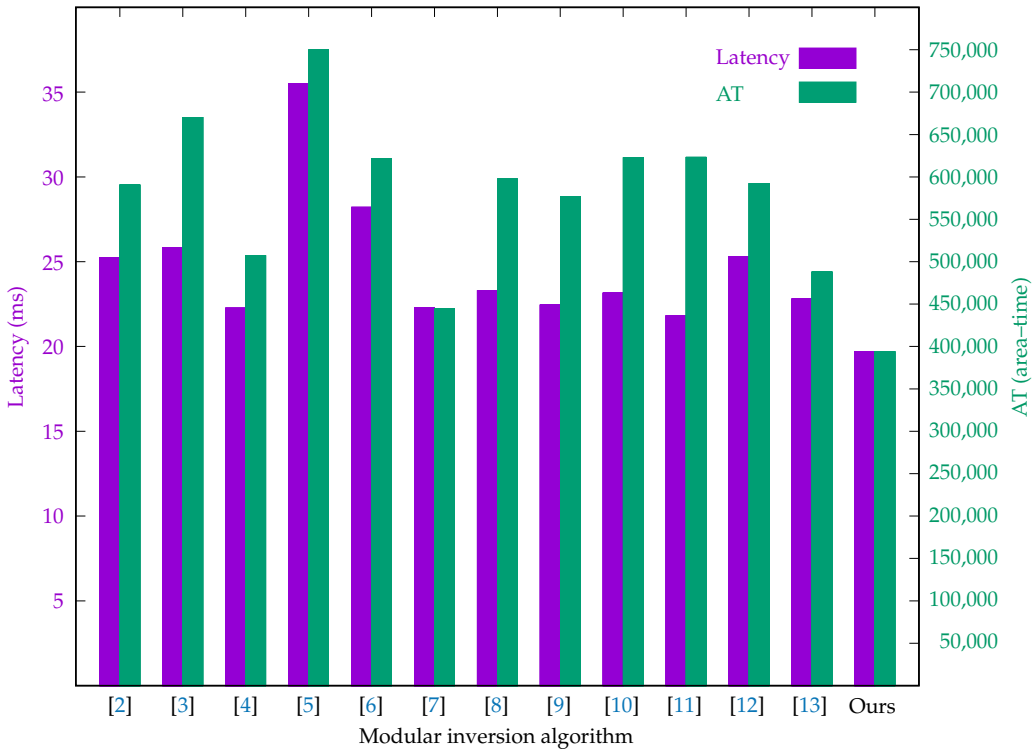

Figure 6. ECC latency and AT comparison of modular inversion algorithms. Details (year and first author's name) of the numbers [*n*] (algorithm) on the horizontal axis are in Table 10.

Algorithm 12, the traditional scalar point multiplication, suffers from side-channel attacks, because its execution time depends on the input scalar d. Side-channel attacks attempt to reveal the secret key from leaked information, such as timing, power consumption, or electromagnetic radiation. We can use the Montgomery ladder algorithm [20] to perform the scalar point multiplication, as shown in Algorithm 13. Its execution takes the same constant time regardless of the input scalar d, making it resistant to side-channel attacks.

Algorithm 13 ScaMulMont (d, P, m, a) (Montgomery Ladder Scalar Point Multiplication).

inputs: $d = \langle 1 d_{n-2} \cdots d_1 d_0 \rangle$ and point $P = [P_x, P_y]$; m and a in $y^2 = x^3 + ax + b \bmod m$
output: $Q = dP$
begin
1 $Q = P$, $R = \mathrm{PD}(P, m, a)$ /* $Q = P$ and $R = 2P$ */
2 **for** $i = n - 2$ **downto** 0 **do**
3 **if** $d_i = 1$
4 $Q = \mathrm{PA}(Q, R, m, a)$ /* $Q = Q + R$ (Algorithm 1) */
5 $R = \mathrm{PD}(R, m, a)$ /* $R = 2R$ (Algorithm 2) */
6 **else**
7 $R = \mathrm{PA}(Q, R, m, a)$ /* $R = Q + R$ (Algorithm 1) */
8 $Q = \mathrm{PD}(Q, m, a)$ /* $Q = 2Q$ (Algorithm 2) */
9 **endfor**
10 **return** Q /* $Q = dP$ */
end

We give the Montgomery ladder scalar point multiplication algorithm's Python code as follows, and give the block diagram of the Montgomery ladder circuit in Figure 7b.

```
def scalar_point_multiplication (P, d, m, a): # scalar point multiplication
    if d == 0: return [-1, -1] # Point O
    Q = P                      # Q =  P
    R = point_doubling (P, m, a) # R = 2P
    for i in range(254, -1, -1): # 254, 253, ..., 2, 1, 0
        if (d >> i) & 1:
            Q = point_addition (Q, R, m, a) # Q = Q + R
            R = point_doubling (R, m, a)    # R = 2R
        else:
            R = point_addition (Q, R, m, a) # R = Q + R
            Q = point_doubling (Q, m, a)    # Q = 2Q
    return Q
```

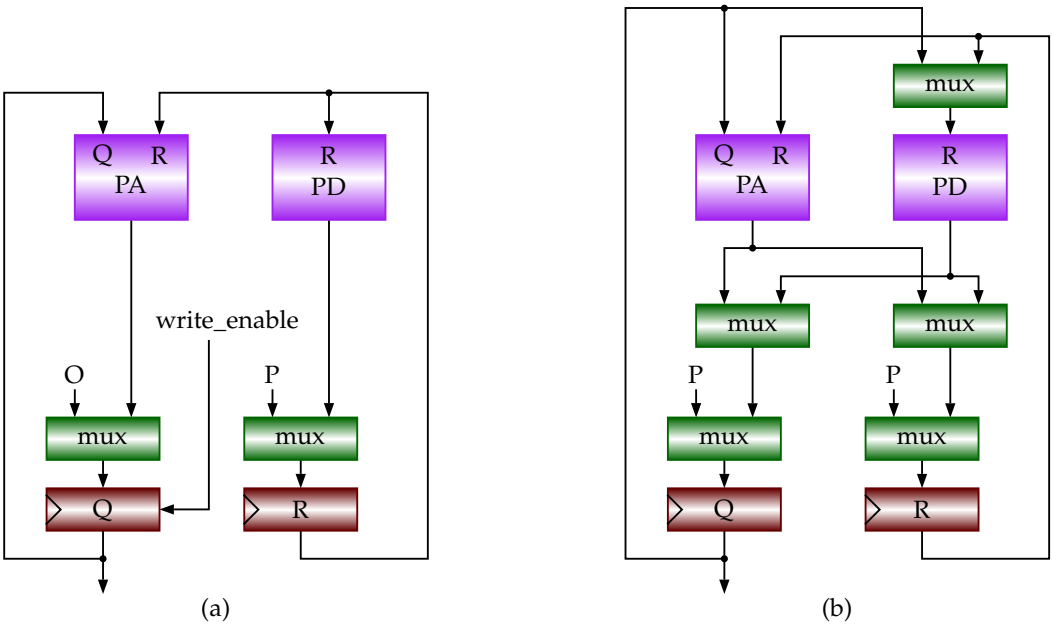

Figure 7. Block diagrams of scalar point multiplication circuits. (**a**) Traditional scalar point multiplication circuit; (**b**) Montgomery ladder scalar point multiplication circuit.

Since multiple PA or PD operations are not performed simultaneously, one PA module and one PD module are sufficient (PA and PD operate in parallel). Note that the iterative

control part based on the scalar d is omitted in the figure. In contrast, the block diagram of the traditional scalar point multiplication circuit is shown in Figure 7a.

Algorithm 13 is resistant to side-channel attacks, while requiring approximately the same execution time and the same hardware resources compared to Algorithm 12, as shown in Table 11. Because the Montgomery ladder scalar point multiplication circuit (Figure 7b) uses more multiplexers, the number of ALMs is larger than Algorithm 12. Algorithm 12 uses a 256-bit register to shift the scalar d, and Algorithm 13 uses a 9-bit counter to control the iterations. Therefore, Algorithm 12 uses more registers than Algorithm 13.

Table 11. Comparison of scalar point multiplication circuits (on Altera Cyclone V FPGA chip).

Algorithm	Cycles	Freq. (MHz)	Latency (ms)	ALMs	Registers	AT
Traditional (Algorithm 12)	317,681	16.13	19.70	12,160	7822	393,546.29
Mont.ladder (Algorithm 13)	318,831	15.86	20.10	12,723	7577	408,087.60

5. Discussion of Design Issues

In the previous sections, we demonstrated through simulations using ModelSim and implementations with the Quartus II EDA tool that the proposed high-radix modular inversion algorithm works correctly and is efficient in terms of execution time and hardware costs. We also showed that the ECC implementation using the proposed algorithm outperformed implementations using other investigated algorithms.

The modular inversion algorithm repeatedly performs addition, subtraction, and shift operations on the variables u, v, x, and y. These variables are typically implemented using registers that are updated on the rising edge of a clock signal. The clock frequency of the circuit is determined by the operation delay between two successive clock rising edges (a clock cycle). Increasing the clock frequency requires decreasing the delay of operations within a clock cycle.

We can use a finite state machine to divide sequential computations into multiple steps and store the results of the steps in registers. This reduces the latency within one clock cycle and increases the clock frequency. However, implementing a finite state machine requires more clock cycles and more registers.

Multiplexers have much lower latency and lower hardware cost compared to carry propagate adders. To reduce hardware costs, if some additions have the same input, instead of using adders and then a multiplexer, we can use a multiplexer before an adder. For example, instead of $s = \text{ mux } (a+b, a+c, a+d, a+e)$, we can have $s = a + \text{ mux } (b, c, d, e)$, which reduces the number of adders. The circuit for calculating txy in Figure 1 is designed in this way to generate tz using a multiplexer before the adder. In addition, to reduce hardware costs, we only use addition for both addition and subtraction calculations. For example, in Figure 1 we only perform the addition $txy = tx + ty + tz$. For radix-8 modular operations, the subtractions $-2m$ and $-3m$ are replaced by $+6m$ and $+5m$, respectively.

The carry-select adder (CSLA) can be used to reduce the latency of the carry propagate adder. For an n bits carry propagate adder, we split the n bits into two $n/2$ bits, the upper $n/2$ bits and the lower $n/2$ bits. The addition of the upper $n/2$ bits is performed simultaneously by two adders, assuming that the carry-in of one adder is 0 and the carry-in of the other adder is 1. Three $n/2$-bit adders (including one for the lower $n/2$ bits) operate in parallel. When the carry-out of the lower $n/2$-bit adder is available, a multiplexer is used to select the correct result from the upper two adders. Using a CSLA by dividing n bits into two $n/2$ bits can reduce the latency by approximately half. On the other hand, the hardware cost increases by more than 50% (one extra $n/2$-bit adder and one $n/2$-bit multiplexer are required). In general, splitting n bits into m (n/m)-bits reduces the latency by about a factor of m, but increases the hardware cost exponentially.

The use of a carry-save adder (CSA) can significantly reduce latency. There is no ripple carry between bits. The result is represented as a carry set and a sum set. A single carry-save adder is equivalent to a 1-bit full adder, which has a low latency. Because of the

representation of the two sets, it is not possible to know the final addition result and its sign (negative or positive) without performing an additional addition, with a carry look-ahead adder for example. Therefore, a CSA is commonly used for intermediate calculations. It takes three sets of inputs and produces two sets of outputs.

As shown in Algorithm 11, our proposed radix-8 modular inversion algorithm allows for arbitrary bit primes, because we considered all combinations of the least significant three bits of prime numbers, as shown in Table 5. If we use a fixed prime m, defined by Secp256k1 [16] for instance, the circuit can be simplified by removing the parts of prime number m whose least significant three bits are not 111. Furthermore, $3m$, $5m$, and $6m$ can be calculated and stored in a constant table in advance. Then, the hardware can use them directly, without any calculations. This speeds up the radix-8 modular inversion calculations.

Our radix-8 modular inversion algorithm allows any bit prime and has no special requirements on the prime, so we can easily use different elliptic curves with the same or different prime sizes, as shown in Table 8. In addition, the hardware implementation of the algorithm is provided in Verilog HDL, which does not rely on special circuit libraries, allowing the algorithm to be implemented on a variety of different platforms.

This paper mainly focused on the modular inversion algorithm and the hardware implementation. We presented the performance and hardware cost of simple low-cost ECC implementations using different modular inversion algorithms, to demonstrate the benefits of the proposed algorithm. To make a fair comparison between implementations, all ECC hardware circuits were identical, except for the modular inversion circuit. From Table 10, we can see that the frequencies of all ECC implementations were quite low. To increase the clock frequency, we can also use pipeline techniques to divide computations into multiple stages and use pipeline registers to store intermediate results. However, the hardware costs will increase due to the use of pipeline registers.

6. Concluding Remarks

In this paper, we proposed a mixed radix-8 modular inversion algorithm and hardware implementation based on 256-bit primes in Verilog HDL and compared its cost performance with other implementations on the Altera Cyclone V FPGA chip. The algorithm and its hardware implementation were area–time efficient with an AT factor of 8.30, which outperformed the other algorithms and implementations. We showed that our algorithm also demonstrates scalability to other prime sizes and adaptability to other cryptographic curves. We also presented the cost performance of an ECC implementation using the proposed modular inversion algorithm. Implementation results also showed that our algorithm reduces the execution time and requires fewer hardware resources than the other investigated algorithms. We presented an efficient implementation of the Montgomery ladder scalar point multiplication algorithm that is resistant to side-channel attacks.

Future work could include shortening the critical path and using carry-select adders and carry-save adders to speed up the addition of large operands. In addition, using a fixed prime m, Secp256k1 for example, we could simplify the circuit by considering only the case where the least significant three bits are equal to 111 and using precomputations of $3m$, $5m$, and $6m$ to speed up the radix-8 modular inversion calculations.

Another important future work is to minimize the latency of the ECC implementation by using longer pipelines. The pipeline stages could be split, so that a modular addition or subtraction can be completed within a single pipeline stage.

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

Conflicts of Interest: The author declares no conflict of interest.

Appendix A. EEA-Based Modular Inversion Algorithms in Python

In the previous sections, we described Algorithms 4–10 (modinv1 to modinv7) and the radix-8 modular inversion algorithm in pseudocode. This appendix provides their implementation codes in Python. We simply summarize the codes as below: modinv1 is the fundamental EEA code; modinv2 removes the division; modinv3 reduces the number of iterations; modinv4 repeatedly divides u and v by two; modinv5 divides $u - v$ or $v - u$ by two; modinv6 assigns tu and tv, so that $tu - tv$ is even; modinv7 uses negative assignments to v and y; and modinv_radix8 gives the code of the proposed algorithm.

```
from random import SystemRandom # random number generator
rand = SystemRandom ()   # strong random number generator
def modinv1 (b, a, m): # return (b * a^{-1}) mod m --------------- fundamental EEA
    u, v = a, m
    x, y = b, 0
    while v != 0:
        q = u // v
        u, v = v, u - q * v
        x, y = y, x - q * y
    return x % m
def modinv2 (b, a, m): # return (b * a^{-1}) mod m -------------- removed division
    u, v = a, m
    x, y = b, 0
    while v != 0:
        q = 0 if u < v else 1
        u, v = v, u - q * v
        x, y = y, x - q * y
    return x % m
def modinv3 (b, a, m): # return (b * a^{-1}) mod m ------------ reduced iterations
    u, v = a, m
    x, y = b, 0
    while u != 1 and v != 1:
        if  u < v: v, y = v - u, y - x
        else:      u, x = u - v, x - y
    if  u == 1: return x % m
    else:       return y % m
def modinv4 (b, a, m): # return (b * a^{-1}) mod m ------ u/2, v/2, Algorithm 2.22
    u, v = a, m
    x, y = b, 0
    while u != 1 and v != 1:
        while u & 1 == 0:
            u = u                       // 2
            if  x & 1 == 0: x =  x      // 2
            else:           x = (x + m) // 2
        while v & 1 == 0:
            v = v                       // 2
            if  y & 1 == 0: y =  y      // 2
            else:           y = (y + m) // 2
        if  u < v: v, y = v - u, y - x
        else:      u, x = u - v, x - y
    if  u == 1: return x % m
    else:       return y % m
def modinv5 (b, a, m): # return (b * a^{-1}) mod m -------------- (u-v)/2, (v-u)/2
    u, v = a, m
    x, y = b, 0
    while u != 1 and v != 1:
        while u & 1 == 0:
            u = u                       // 2
            if  x & 1 == 0: x =  x      // 2
            else:           x = (x + m) // 2
        while v & 1 == 0:
            v = v                       // 2
            if  y & 1 == 0: y =  y      // 2
            else:           y = (y + m) // 2
        if  u < v:
            v, y = (v - u) // 2, y - x
```

```
            if  y & 1 == 0: y =  y      // 2
            else:           y = (y + m) // 2
        else:
            u, x = (u - v) // 2, x - y
            if  x & 1 == 0: x = x       // 2
            else:           x = (x + m) // 2
    if  u == 1: return x % m
    else:       return y % m
def modinv6 (b, a, m): # return (b * a^{-1}) mod m --- no u/2, no v/2, multiplexer
    u, v = a, m
    x, y = b, 0
    while u != 1 and v != 1:
        if u & 1 == 1: tv, ty = v, y
        else:          tv, ty = 0, 0
        if v & 1 == 1: tu, tx = u, x
        else:          tu, tx = 0, 0
        tuv, txy = tu - tv, tx - ty
        uv = tuv                        // 2
        if txy & 1 == 0: xy =  txy      // 2
        else:            xy = (txy + m) // 2
        if  uv < 0: v, y = -uv, -xy
        else:       u, x =  uv,  xy
    if  u == 1: return x % m
    else:       return y % m
def modinv7 (b, a, m): # return (b * a^{-1}) mod m ----------- negative assignment
    u, v = a, -m
    x, y = b, -0
    while u != 1:
        if u & 1 == 1: tv, ty = v, y
        else:          tv, ty = 0, 0
        if v & 1 == 1: tu, tx = u, x
        else:          tu, tx = 0, 0
        tuv, txy = tu + tv, tx + ty
        uv = tuv                        // 2
        if txy & 1 == 0: xy =  txy      // 2
        else:
            if txy < 0:  xy = (txy + m) // 2
            else:        xy = (txy - m) // 2
        if  uv < 0: v, y = uv, xy
        else:       u, x = uv, xy
    if  x < 0: x = x + m
    return x
def modinv_radix8 (b, a, m): # return (b * a^{-1}) mod m # proposed radix-8 modinv
    u, v = a, -m
    x, y = b, -0
    while u != 1:
        if u & 1 == 1: tv, ty = v, y
        else:          tv, ty = 0, 0
        if v & 1 == 1: tu, tx = u, x
        else:          tu, tx = 0, 0
        tuv, txy = tu + tv, tx + ty          # tuv is even
        if tuv & 6 == 0:                     # radix 8:
            uv = tuv                                          // 8
            if txy & 1 == 0:
                if txy & 2 == 0:
                    if txy & 4 == 0:         xy =  txy        // 8
                    else:                    xy = (txy + 4 * m) // 8
                else:
                    if txy & 4 == (m*2 & 4): xy = (txy - 2 * m) // 8
                    else:                    xy = (txy + 2 * m) // 8
            else:
                if txy & 6 == m & 6:         xy = (txy -     m) // 8
                else:
                    if txy & 2 == m & 2:     xy = (txy + 3 * m) // 8
                    else:
                        if txy & 4 != m & 4: xy = (txy +     m) // 8
                        else:                xy = (txy - 3 * m) // 8
```

```
        else:
            if tuv & 2 == 0:                # radix 4:
                uv = tuv                                        // 4
                if txy & 1 == 0:
                    if txy & 2 == 0:        xy =  txy           // 4
                    else:                   xy = (txy + 2 * m) // 4
                else:
                    if txy & 3 == m & 3:    xy = (txy -     m) // 4
                    else:                   xy = (txy +     m) // 4
            else:                           # radix 2:
                uv = tuv                                        // 2
                if txy & 1 == 0:            xy =  txy           // 2
                else:
                    if txy < 0:             xy = (txy +     m) // 2
                    else:                   xy = (txy -     m) // 2
        if  uv < 0: v, y = uv, xy
        else:       u, x = uv, xy
    if  x < 0: x = x + m
    return x
m  = int (0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f)
b  = rand.getrandbits (256) % m
a  = rand.getrandbits (256) % m
c1 = modinv1 (b, a, m)
c2 = modinv2 (b, a, m)
c3 = modinv3 (b, a, m)
c4 = modinv4 (b, a, m)
c5 = modinv5 (b, a, m)
c6 = modinv6 (b, a, m)
c7 = modinv7 (b, a, m)
c  = modinv_radix8 (b, a, m)
print ('b  = 0x{:064x}'.format(b))
print ('a  = 0x{:064x}'.format(a))
print ('m  = 0x{:064x}'.format(m))
print ('c1 = 0x{:064x}'.format(c1))
print ('c2 = 0x{:064x}'.format(c2))
print ('c3 = 0x{:064x}'.format(c3))
print ('c4 = 0x{:064x}'.format(c4))
print ('c5 = 0x{:064x}'.format(c5))
print ('c6 = 0x{:064x}'.format(c6))
print ('c7 = 0x{:064x}'.format(c7))
print ('c  = 0x{:064x}'.format(c))
assert c1 == c2 == c3 == c4 == c5 == c6 == c7 == c
assert (c * a) % m == b # verify correctness
```

The last `assert` statement verifies the correctness of the calculated modular inversion result `c`. Below is an example of the output when the code is executed. We can see that for the same inputs b, a, m, the eight functions have the same output. These outputs of the Python code are used to check the correctness of the circuit's outputs. For example, the value of the signal c in Figure 2 is the same as the output `c` of the Python code.

```
$ python3 modinv12345678.py
b  = 0x9cfa1c993911914be0f15bd74a878abe0079c6254b961b82e1abda76387d1d85
a  = 0xd5076ae274e874c2eb0f7778717c39460236549ddd9fc651e68a0c0e787b4ce8
m  = 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f
c1 = 0xe8e5ac2e1d3358894ce1b3342737b38c39b89059dd55d3c4741626de8270228e
c2 = 0xe8e5ac2e1d3358894ce1b3342737b38c39b89059dd55d3c4741626de8270228e
c3 = 0xe8e5ac2e1d3358894ce1b3342737b38c39b89059dd55d3c4741626de8270228e
c4 = 0xe8e5ac2e1d3358894ce1b3342737b38c39b89059dd55d3c4741626de8270228e
c5 = 0xe8e5ac2e1d3358894ce1b3342737b38c39b89059dd55d3c4741626de8270228e
c6 = 0xe8e5ac2e1d3358894ce1b3342737b38c39b89059dd55d3c4741626de8270228e
c7 = 0xe8e5ac2e1d3358894ce1b3342737b38c39b89059dd55d3c4741626de8270228e
c  = 0xe8e5ac2e1d3358894ce1b3342737b38c39b89059dd55d3c4741626de8270228e
```

Appendix B. Elliptic Curve Diffie–Hellman (ECDH) Key Exchange Algorithm in Python

The ECDH algorithm is used to reach key agreement between two parties over an insecure network, as shown in Table 1. The following Python code demonstrates that two parties, Alice and Bob for example, obtain the same secure key by calling scalar point multiplications. We skip the communication process and focus on how to implement scalar point multiplication by calling point addition and point doubling.

```
from random import SystemRandom # random number generator
rand = SystemRandom ()   # strong random number generator
def modadd (a, b, m): # return (a + b) mod m; a, b < m
    s = a + b
    if s > m:
        s = s - m
    return s
def modsub (a, b, m): # return (a - b) mod m; a, b < m
    s = a - b
    if s < 0:
        s = s + m
    return s
def modmul (a, b, m): # return (a * b) mod m; a, b < m # shift-sub (SSMM)
    u, v, s = a, b, 0
    while v != 0:
        if v & 1 == 1:
            s = s + u
            if s > m:
                s = s - m
        v = v >> 1
        u = u << 1
        if u > m:
            u = u - m
    return s
def modinv (b, a, m): # return (b * a^{-1}) mod m # proposed radix-8 modinv
    u, v = a, -m
    x, y = b, -0
    while u != 1:
        if u & 1 == 1: tv, ty = v, y
        else:          tv, ty = 0, 0
        if v & 1 == 1: tu, tx = u, x
        else:          tu, tx = 0, 0
        tuv, txy = tu + tv, tx + ty          # tuv is even
        if tuv & 6 == 0:                     # radix 8:
            uv = tuv                                                  // 8
            if txy & 1 == 0:
                if txy & 2 == 0:
                    if txy & 4 == 0:         xy =  txy                // 8
                    else:                    xy = (txy + 4 * m) // 8
                else:
                    if txy & 4 == (m*2 & 4): xy = (txy + 6 * m) // 8 # -2m
                    else:                    xy = (txy + 2 * m) // 8
            else:
                if txy & 6 == m & 6:         xy = (txy -     m) // 8
                else:
                    if txy & 2 == m & 2:     xy = (txy + 3 * m) // 8
                    else:
                        if txy & 4 != m & 4: xy = (txy +     m) // 8
                        else:                xy = (txy + 5 * m) // 8 # -3m
        else:
            if tuv & 2 == 0:                 # radix 4:
                uv = tuv                                              // 4
                if txy & 1 == 0:
                    if txy & 2 == 0:         xy =  txy                // 4
                    else:                    xy = (txy + 2 * m) // 4
                else:
                    if txy & 3 == m & 3:     xy = (txy -     m) // 4
                    else:                    xy = (txy +     m) // 4
```

```
            else:                               # radix 2:
                uv = tuv                                        // 2
                if txy & 1 == 0:                xy =  txy       // 2
                else:
                    if txy < 0:                 xy = (txy +   m) // 2
                    else:                       xy = (txy -   m) // 2
        if  uv < 0: v, y = uv, xy
        else:       u, x = uv, xy
    if  x < 0: x = x + m
    return x
def point_addition (P, Q, m, a): # point addition R = P + Q
    x1, y1 = P
    x2, y2 = Q
    if x1 == -1 and y1 == -1: return Q # O + Q
    if x2 == -1 and y2 == -1: return P # P + O
    if x1 == x2:
        if modadd (y1, y2, m) == 0: return [-1, -1] # Point O
        else: return point_doubling (P, m, a) # 2P
    # s = ((y1 - y2) / (x1 - x2)) mod m
    s = modinv (modsub (y1, y2, m), modsub (x1, x2, m), m)
    # rx = (s * s - x1 - x2) mod m
    rx = modsub (modmul (s, s, m), modadd (x1, x2, m), m)
    # ry = (s * (x1 - rx) - y1) mod m
    ry = modsub (modmul (s, modsub (x1, rx, m), m), y1, m)
    return [int (rx), int (ry)]
def point_doubling (P, m, a): # point doubling R = 2P
    x, y = P
    if y == 0: return [-1, -1] # Point O
    # s = ((3 * x * x + a) / (2 * y)) mod m
    s = modinv (modadd(a, modmul(modmul(x, x, m), 3, m), m), modadd(y, y, m), m)
    # rx = (s * s - 2 * x) mod m
    rx = modsub (modmul (s, s, m), modmul (x, 2, m), m)
    # ry = (s * (x - rx) - y) mod m
    ry = modsub (modmul (s, modsub (x, rx, m), m), y, m)
    return [int (rx), int (ry)]
def scalar_point_multiplication (P, d, m, a): # scalar point multiplication
    if d == 0: return [-1, -1] # Point O
    k = d
    Q = [-1, -1] # Point O
    R = P
    while k != 0:
        if k & 1:
            Q = point_addition (Q, R, m, a) # Q = Q + R
        R = point_doubling (R, m, a)        # R = 2R
        k >>= 1
    return Q
a   = int (0x0000000000000000000000000000000000000000000000000000000000000000)
b   = int (0x0000000000000000000000000000000000000000000000000000000000000007)
m   = int (0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f)
x   = int (0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798)
y   = int (0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8)
P   = [x, y] # Elliptic curve Diffie-Hellman (ECDH) key agreement:
da  = rand.getrandbits (256) % m                # Alice's private key
db  = rand.getrandbits (256) % m                # Bob's   private key
Qa  = scalar_point_multiplication ( P, da, m, a) # Alice's public  key
Qb  = scalar_point_multiplication ( P, db, m, a) # Bob's   public  key
Qab = scalar_point_multiplication (Qb, da, m, a) # Alice calculates shared key
Qba = scalar_point_multiplication (Qa, db, m, a) # Bob   calculates shared key
print ('da   = 0x{:064x}'.format(da),     end=' ')
print ('db   = 0x{:064x}'.format(db))
print ('Qax  = 0x{:064x}'.format(Qa[0]),  end=' ')
print ('Qay  = 0x{:064x}'.format(Qa[1]))
print ('Qbx  = 0x{:064x}'.format(Qb[0]),  end=' ')
print ('Qby  = 0x{:064x}'.format(Qb[1]))
print ('Qabx = 0x{:064x}'.format(Qab[0]), end=' ')
print ('Qaby = 0x{:064x}'.format(Qab[1]))
print ('Qbax = 0x{:064x}'.format(Qba[0]), end=' ')
```

```
print ('Qbay = 0x{:064x}'.format(Qba[1]))
assert (Qa [1] * Qa [1]) % m == (Qa [0] * Qa [0] * Qa [0] + a * Qa [0] + b) % m
assert (Qb [1] * Qb [1]) % m == (Qb [0] * Qb [0] * Qb [0] + a * Qb [0] + b) % m
assert (Qab[1] * Qab[1]) % m == (Qab[0] * Qab[0] * Qab[0] + a * Qab[0] + b) % m
assert (Qba[1] * Qba[1]) % m == (Qba[0] * Qba[0] * Qba[0] + a * Qba[0] + b) % m
assert Qab == Qba # verify correctness
```

Python functions for modular addition (`modadd`), modular subtraction (`modsub`), modular multiplication (`modmul`), and modular inversion (`modinv`) are provided. These functions are used by point addition (`point_addition`) and point doubling (`point_doubling`). All the codes are hardware-oriented for circuit design.

The last `assert` statement is used to check whether the two parties obtained the same shared secure key. Below is an example of the output when the code is executed. We can see that `Qbax` is equal to `Qabx` (shared secure key). These outputs of the Python code are used to check the correctness of the circuit's outputs. For example, the values of signals qx and qy in Figure 5 are the same as the outputs `Qax` and `Qay`, respectively, of the Python code.

```
$ python3 ecdh.py
da   = 0x650aa7095daeaa37ab9051541f0ce304f8969a6d88bb3bebb4fe680fca9a2595
db   = 0xedc68f194c4e30d6ef90467df822b00e5ef122dea48c9d1c54817080d1a341f4
Qax  = 0x167d2537aa6bbd8d978b58be0f9466520b7b184e205ff96a9ff567b35b32c7b7
Qay  = 0xde3961553d36551f92726fee0e332133960edddccd2784b98b2af730d2fc6e14
Qbx  = 0x839da64a414c2243a5526230603109be9c615613a9e98c3d650bb0488580bbda
Qby  = 0x96e88e99304a5afcdd77c4f3b3327a28162627ebe08194baa0c78dfb67a11042
Qabx = 0x1f254c7da15899275cdcab9d992f58251a4ab630fe9864d20cf317ab57749947
Qaby = 0xd6cb400b3c49d33d3df28f9d34fa09f8b6c8edf117a378c5a45d0a51e6c0debc
Qbax = 0x1f254c7da15899275cdcab9d992f58251a4ab630fe9864d20cf317ab57749947
Qbay = 0xd6cb400b3c49d33d3df28f9d34fa09f8b6c8edf117a378c5a45d0a51e6c0debc
```

References

1. Burton, D. *The History of Mathematics/An Introduction*, 7th ed.; McGraw-Hill: New York, NY, USA, 2011. [CrossRef]
2. Hankerson, D.; Menezes, A.; Vanstone, S. *Guide to Elliptic Curve Cryptography*; Springer: New York, NY, USA, 2004. [CrossRef]
3. Hossain, M.S.; Kong, Y. High-Performance FPGA Implementation of Modular Inversion over F_256 for Elliptic Curve Cryptography. In Proceedings of the 2015 IEEE International Conference on Data Science and Data Intensive Systems, Sydney, Australia, 11–13 December 2015; pp. 169–174. [CrossRef]
4. Daly, A.; Marnane, W.; Kerins, T.; Popovici, E. Division in GF(p) for Application in Elliptic Curve Cryptosystems on Field Programmable Logic. In *New Algorithms, Architectures and Applications for Reconfigurable Computing*; Springer: Boston, MA, USA, 2005; pp. 219–229. [CrossRef]
5. Mrabet, A.; El-Mrabet, N.; Bouallegue, B.; Mesnager, S.; Machhout, M. An efficient and scalable modular inversion/division for public key cryptosystems. In Proceedings of the 2017 International Conference on Engineering & MIS (ICEMIS), Monastir, Tunisia, 8–10 May 2017; pp. 1–6. [CrossRef]
6. Chen, C.; Qin, Z. Fast Algorithm and Hardware Architecture for Modular Inversion in GF(p). In Proceedings of the 2009 Second International Conference on Intelligent Networks and Intelligent Systems, Tianjin, China, 1–3 November 2009; pp. 43–45. [CrossRef]
7. Choi, P.; Lee, M.K.; Kong, J.T.; Kim, D.K. Efficient Design and Performance Analysis of a Hardware Right-shift Binary Modular Inversion Algorithm in GF(p). *J. Semicond. Technol. Sci.* **2017**, *17*, 425–437. [CrossRef]
8. Wang, D.; Lin, Y.; Hu, J.; Zhang, C.; Zhong, Q. FPGA Implementation for Elliptic Curve Cryptography Algorithm and Circuit with High Efficiency and Low Delay for IoT Applications. *Micromachines* **2023**, *14*, 1037. [CrossRef] [PubMed]
9. Yang, D.; Dai, Z.; Li, W.; Chen, T. An Efficient ASIC Implementation of Public Key Cryptography Algorithm SM2 Based on Module Arithmetic Logic Unit. In Proceedings of the 2019 IEEE 13th International Conference on ASIC (ASICON), Chongqing, China, 29 October–1 November 2019; pp. 1–4. [CrossRef]
10. Yan, X.; Li, S. Modified modular inversion algorithm for VLSI implementation. In Proceedings of the 2007 7th International Conference on ASIC, Guilin, China, 22–25 October 2007; pp. 90–93. [CrossRef]
11. Dong, X.; Zhang, L.; Gao, X. An Efficient FPGA Implementation of ECC Modular Inversion over F256. In Proceedings of the 2nd International Conference on Cryptography, Security and Privacy, Guiyang, China, 16–18 March 2018; pp. 29–33. [CrossRef]

12. Hao, Y.; Zhong, S.; Ma, M.; Jiang, R.; Huang, S.; Zhang, J.; Wang, W. Lightweight Architecture for Elliptic Curve Scalar Multiplication over Prime Field. *Electronics* **2022**, *11*, 2234. [CrossRef]
13. Guo, K.Y.; Fang, W.C.; Fahier, N. An Efficient Hardware Design of Prime Field Modular Inversion/Division for Public Key Cryptography. In Proceedings of the 2023 IEEE International Symposium on Circuits and Systems (ISCAS), Monterey, CA, USA, 21–25 May 2023; pp. 1–5. [CrossRef]
14. Koblitz, N. Elliptic curve cryptosystems. *Math. Comput.* **1987**, *48*, 203–209. Available online: https://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866109-5/S0025-5718-1987-0866109-5.pdf (accessed on 11 October 2024). [CrossRef]
15. Miller, V.S. Use of Elliptic Curves in Cryptography. In *Proceedings of the Advances in Cryptology—CRYPTO '85 Proceedings*; Springer: Berlin/Heidelberg, Germany, 1986; pp. 417–431. Available online: https://link.springer.com/content/pdf/10.1007/3-540-39799-X_31.pdf?pdf=inline%20link (accessed on 10 October 2024).
16. Certicom Corp. Standards for Efficient Cryptography. SEC 2: Recommended Elliptic Curve Domain Parameters. 2010. Available online: http://www.secg.org/sec2-v2.pdf (accessed on 10 October 2024).
17. Barker, E.; Chen, L.; Roginsky, A.; Vassilev, A.; Davis, R. *SP 800-56A Rev. 3, Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography*; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2018. Available online: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar3.pdf (accessed on 10 October 2024).
18. Li, Y. Hardware Implementations of Elliptic Curve Cryptography Using Shift-Sub Based Modular Multiplication Algorithms. *Cryptography* **2023**, *7*, 1–29. [CrossRef]
19. Li, Y.; Chu, W. Shift-Sub Modular Multiplication Algorithm and Hardware Implementation for RSA Cryptography. In *Proceedings of the 17th International Conference on Information Assurance and Security, Lecture Notes in Networks and Systems*; Springer: Cham, Switzerland, 2021; pp. 541–552. [CrossRef]
20. Oliveira, T.; López, J.; Rodríguez-Henríquez, F. The Montgomery ladder on binary elliptic curves. *J. Cryptogr. Eng.* **2018**, *8*, 241–258. [CrossRef]