

Article

Not peer-reviewed version

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

[Yamin Li](#) *

Posted Date: 11 September 2024

doi: 10.20944/preprints202409.0891.v1

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




Preprints.org is a free multidiscipline platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

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

Abstract: Modular inversion on large operands is a time-consuming calculation used in elliptic curve cryptography. Its hardware implementation requires extensive hardware resources such as lookup tables and registers. We investigate 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 short execution time and low hardware cost. We present a detailed radix-8 hardware implementation based on 256-bit primes in Verilog HDL and compare its cost performance with other implementations. Our implementation on the Altera Cyclone V FPGA chip uses 1227 ALMs (Adaptive logic modules) and 1037 registers. The modular inversion calculation takes 3.67 microseconds. The AT (Area time) factor is 8.30, outperforming other implementations. We also present an implementation of elliptic curve cryptography using the proposed radix-8 modular inversion algorithm. The implementation results also show that our modular inversion algorithm is more efficient in area time than other algorithms.

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

1. Introduction

Modular inversion is an important computation in elliptic curve cryptography (ECC). ECC provides a secure key agreement between two parties over an insecure network. It 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 to 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 mainly two 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 longer time and requires more registers than EEA. Therefore, we will focus our design on using the EEA.

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

For calculating $r = a^{-1} \bmod m$, EEA first initializes u, v, x, y with $a, m, 1, 0$, respectively. Then 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 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 shift. This ensures that the value being right-shifted is even since the prime number m is odd. Next, if $u \geq v$, u and x will be replaced by $u - v$ and $x - y$, respectively. Otherwise, v and y will be 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 $y - x$ needs also to be divided by two: If $x - y$ or $y - x$ is odd, m is added before the division. Division by two is done 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 does, they perform $u + v$ for new u or v . This operation slows down the speed at which u or v reaches 1, increasing execution time. Chen and Qin proposed a modular inversion algorithm [6] that uses only 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. Also, 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 are 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 a sequential condition checking for the calculations 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] gave 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.

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 short execution time and small hardware resources. We give its detailed hardware implementation in Verilog HDL based on 256-bit primes. It has lower hardware costs for ALMs and registers and has better performance than other algorithms. The implementation on the Altera Cyclone V FPGA chip uses 1227 ALMs and 1037 registers and takes 3.67 microseconds for the modular inversion computation. It achieves an AT factor of 8.30, lower than all other implementations. We also implement ECC using different modular inversion algorithms and compare their cost performance.

The rest of the paper is organized as follows. Section 2 introduces ECC and modular inversion algorithms. Instead of pseudocodes, all modular inversion algorithms are provided in Python code. Section 3 proposes a mixed radix-8 modular inversion algorithm, gives its hardware implementation in Verilog HDL, and compares its cost performance with other algorithms. Section 4 provides an ECC implementation using the proposed algorithm and compares its cost performance with the ECC implementations using other modular inversion algorithms. And Section 5 concludes the paper.

2. ECC and Modular Inversion Algorithms

This section briefly introduces elliptic curve cryptography and modular inversion algorithms based on the extended Euclidean algorithm.

2.1. Elliptic Curve Cryptography

ECC [12,13] 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 equation:

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

For example, Secp256k1 [14] 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 = 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f
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 [14,15]. The ECDH protocol is shown in Table 1.

Table 1. Elliptic curve Diffie–Hellman 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	Generate a secret d_b
Calculate $Q_a = d_a P$	Calculate $Q_b = d_b P$
Expose Q_a	Expose Q_b
Get Q_b from Bob	Get Q_a from Alice
Calculate $Q_{ab} = d_a Q_b$	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 ($Q_{abx} = Q_{bax}$).

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

```
da = 0x0e26233af432c34fa2523e72b64ebcf5abb4f7cb07d8f25909160a50f584f461
qax = 0x672751a7b8ec03f4610c364d3776fc5200f401aa26074b10d33c2c202fc63330
qay = 0xdbf4a0758d81414d2ee049a07c3d8288428235dec9dbfb7fb15d6f478cb4ccf6
```

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

```
db = 0xbbb481621f91b8e9225649109919a7962cec4225c981b6b68b4f8d8a356cc6d9
qbx = 0x09aff71612598a88d5299ae81e3161a2c9045f343315da8cfabb4dc55253041f
qby = 0x736b118369583448c0b77eed11ac26af31450d406130d82a1bc56cc3a1835b1d
```

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

```
qabx = 0x0f675b3195fd6a6f06c9a6960ff2a4f647f637f513c8bb7bedc8a89311f62df2
qaby = 0x79de922da4db277fe0c674277243c1dfd0653913d037fb07e955c3cdf21e69c7
```

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

Qbax = 0x0f675b3195fd6a6f06c9a6960ff2a4f647f637f513c8bb7bedc8a89311f62df2

Qbay = 0x79de922da4db277fe0c674277243c1dfd0653913d037fb07e955c3cdf21e69c7

Now, Alice and Bob have the same secret key ($Q_{abx} = Q_{bax}$). 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 $Q = dP$ calls 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 .

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

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 = 0xf7c7dafb820a20da1a73c36465f2fe37bfd98ce4ef3a10a5df110abda03b20a3d
Py = 0xa442a2d1b8bde4a09e45725add5daae89e726b56f0e8fe6609dacf5279b2564
Qx = 0xe106c069450b2663febb83e29b67fa93c4c48a45d5fbe7ce4ddb8ceb601fcc1d
Qy = 0xc9da9bd440909c8862c06a44d432d2dd45284636b7049b9bf4695f9e4018d2f2
Rx = 0xfd52a0334e16f8cf45a6b0820887a9e8b1b180516a76c8adfe95df98aeef376
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 .

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

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).

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

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 = 0x5491ee8b73a4ed9713ed32e467de5100b80861babf8fffd09fd595ab457d042c9
Ry = 0xf91e6a4e132a1bdf4f5c846559431ec7373de8872b719f188b5902932f0a2b30
```

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

2.3. Modular Inversion Algorithms

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

$$r = a^{-1} \bmod m \quad (4)$$

That is, $ra = 1 \bmod m$. The Python code below implements the modular inversion calculation using Fermat's Little Theorem. When executed, it outputs 4 4 4. The first output value is calculated by the code, and the rest are for checking. This calculation consists of costly modular multiply and modular squaring, very similar to RSA exponentiation [16].

```
# Fermat's Little Theorem, a^{-1} = a^{m-2} mod m
def modinv (a, m): # fermat.py: 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(a, m), pow(a, -1, m), pow(a, m-2, m))
```

The extended Euclidean algorithm can be used for the modular inversion calculation. Below is the fundamental extended Euclidean algorithm given in Python code (modinv_algo_1.py), where q is the integer quotient of u divided by v .

```
def modinv (b, a, m): # modinv_algo_1.py: return b * a^{-1} mod m
    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
    if u == 1: return x % m
    else: return 0; # a is not invertible.
print(modinv(1, 3, 11))
```

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

An example of running modinv_algo_1.py using modinv(1, 3, 11) is shown in Table 2 ($b = 1$, $a = 3$, and $m = 11$). The calculation finishes when $v = 0$. Because $u = 1$, the result $a^{-1} \bmod m = x \bmod m = 4 \bmod 11 = 4$. 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 modinv_algo_1.py: print(modinv(1, 3, 11)). It calculates $r = 3^{-1} \bmod 11$. The result is $x \bmod 11 = 4$.

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. We can remove division by setting the quotient to 0 or 1, as shown below (modinv_algo_2.py).

```
def modinv (b, a, m): # modinv_algo_2.py: return b * a^{-1} mod m
    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
if u == 1: return x % m
else:     return 0; # a is not invertible.

```

The algorithm yields a quotient of 0 or 1 based on the comparison of u and v . If the quotient is a 1, a subtraction is performed. Otherwise, no calculation is performed, resulting in a slow calculation speed. The calculation of $\text{modinv}(1, 3, 11)$ will require 9 iterations. We can modify the algorithm as follows (`modinv_algo_3.py`). This reduces the number of iterations by about half.

```

def modinv(b, a, m): # modinv_algo_3.py: return b * a^{-1} mod m
    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

```

We can check u first before the subtractions. If it is even, we can shift it to the right by one bit (the least significant bit 0 is shifted out). Correspondingly, x must also be shifted. To ensure the shifted value is even, m will be added to x before the shift if x is odd. Note that m is odd because it is a prime number. This shift of u and x can be performed repeatedly until u becomes an odd number. Similar actions can be applied to v and y . Then we can have an algorithm as follows (`modinv_algo_4.py`). In fact, this is Algorithm 2.22 provided in [2] and implemented in Verilog HDL in [17].

```

def modinv(b, a, m): # modinv_algo_4.py: return b * a^{-1} mod m
    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

```

When the two inner while loops 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. The algorithm is shown below (`modinv_algo_5.py`).

```

def modinv(b, a, m): # modinv_algo_5.py: return b * a^{-1} mod m
    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

```

The two inner while loops can be replaced by assigning u , x , v , y , or 0 to the temporary variables tu , tx , tv , and ty . The following assignments make $tu - tv$ even, so we can shift it to the right by one bit: (1) If both u and v are odd, those temporary variables are assigned with u , x , v , y , respectively. (2) If u is even and v is odd, those temporary variables are assigned with u , x , 0, 0, respectively. (3) If u is odd and v is even, those temporary variables are assigned with 0, 0, v , y , respectively. Such elimination shortens the latency from the carry propagation adder to the multiplexer and speeds up the calculation. The algorithm is shown below (modinv_algo_6.py).

```

def modinv (b, a, m): # modinv_algo_6.py: return b * a^{-1} mod m
    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

```

The algorithm above 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. That is, $u = u - v = u + (-v)$ becomes $u = u + v$, and $x = x - y = x + (-y)$ becomes $x = x + y$ with the negative assignments to v and y . Similarly, $v = -(u - v)$ becomes $v = u + v$, and $y = -(x - y)$ becomes $y = x + y$ with the negative assignments to v and y . Therefore, $u + v$ and $x + y$ are sufficient for the calculation. The algorithm is given below (modinv_algo_7.py). 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, no adjustment of $x = x - m$ or $x = x \bmod m$ is required.

```

def modinv (b, a, m): # modinv_algo_7.py: return b * a^{-1} mod m
    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
if txy & 1 == 0: xy = txy // 2
else:
    if tx < 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

```

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 shown below (modinv_radix_8.py). To calculate $r = a^{-1} \bmod m$, we initialize $u = a$ and $v = -m$ 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. If the three least significant bits of tuv are 000, it is shifted to the right by three bits (radix-8). Otherwise, if the two least significant 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.

```

def modinv (b, a, m): # modinv_radix_8.py: return b * a^{-1} mod m
    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
        if tuv & 6 == 0:
            uv = tuv
            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:
                uv = tuv
                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:
                uv = tuv
                if txy & 1 == 0: xy = txy // 2
                else:

```

```

        if tx < 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
```

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 shifting. Table 3 lists such adjustments based on the three least significant bits of txy and the three least significant bits of m for the radix-8 operations, where x represents a don't care. The three least significant bits of the adjusted value are 000, as shown in the comment column of the table.

Table 3. XY adjustment for shift right by three-bit 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 4 lists the adjustments based on the two least significant bits of txy and the two least significant bits of m for the radix-4 operations. The two least significant bits of the adjusted value are 00, as shown in the comment column of the table.

Table 4. XY adjustment for shift right by two-bit 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		$(txy + m) // 4$	$3 + 1 = 4$

An example of $c = b \times a^{-1} \bmod m$ is shown below. The `modinv_radix_8` algorithm takes 206 iterations to reach $u = 1$ and $v = -1$. In contrast, the `modinv_radix_4` and `modinv_radix_2` algorithms require 243 and 356 iterations, respectively.

```
b = 0x9cfa1c993911914be0f15bd74a878abe0079c6254b961b82e1abda76387d1d85
a = 0xd5076ae274e874c2eb0f7778717c39460236549ddd9fc651e68a0c0e787b4ce8
m = 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffeffffc2f
c = 0xe8e5ac2e1d3358894ce1b3342737b38c39b89059dd55d3c4741626de8270228e
```

To reduce the number of adders, we use multiplexers to select an appropriate value and assign it to the temporary variable tz . And then we perform $txy = tx + ty + tz$. Based on the three least significant bits of tuv , we assign $txy \gg 1$, $txy \gg 2$, or $txy \gg 3$ (shift right) to xy . Figure 1 shows the block diagram of the proposed mixed radix-8 modular inversion circuit. Because we perform addition for $txy = tx + ty + tz$, $-2m$ and $-3m$ are replaced by $+6m$ and $+5m$, respectively. Also, 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 registers u , v , x , and y are shown with red rectangles. The multiplexers are drawn with green rectangles. The blue rectangles are adders. 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.

$$c = b \times a^{-1} \bmod m$$

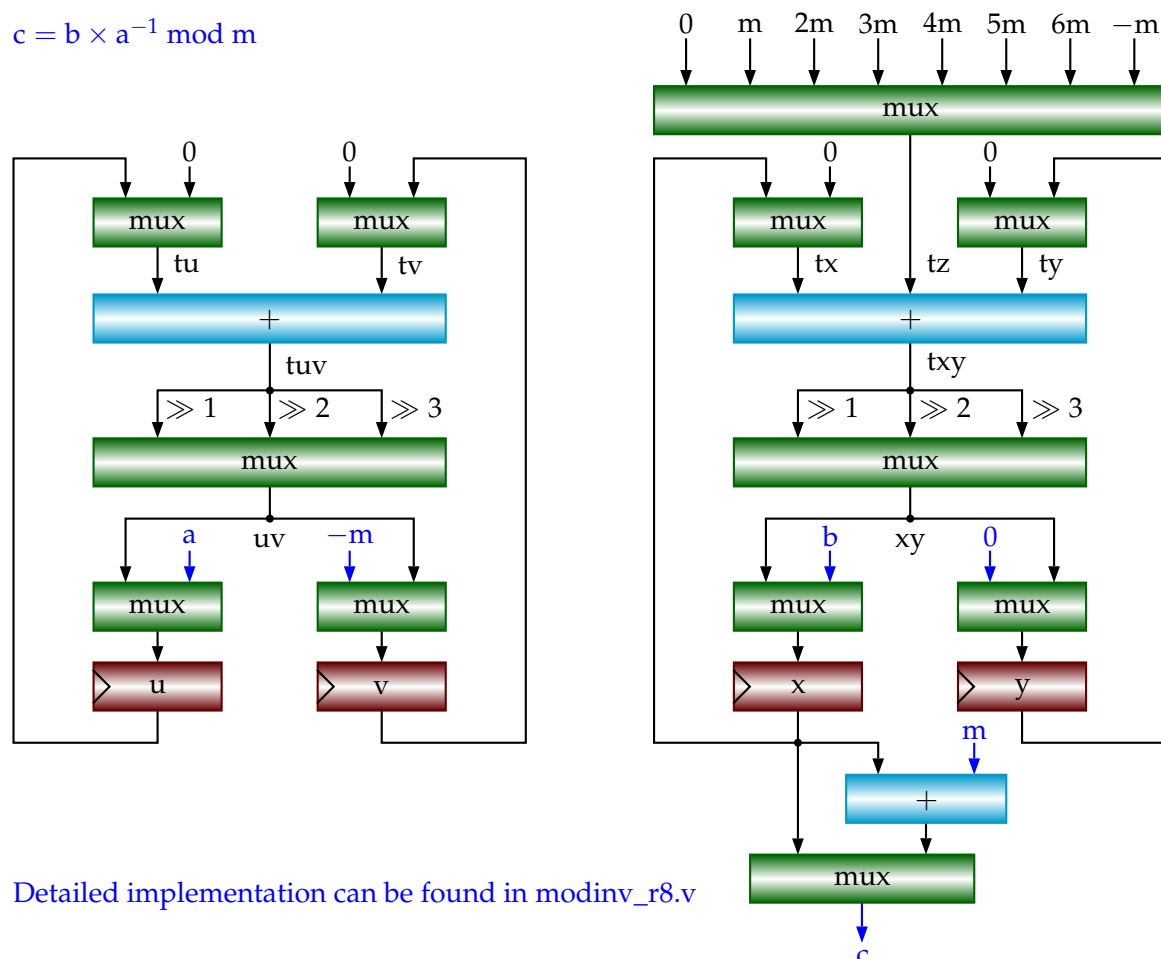


Figure 1. Block diagram of the proposed mixed radix-8 modular inversion circuit.

Below we give the hardware implementation code in Verilog HDL for the proposed mixed 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 Secp256k1 elliptic curve, the input and output signals b , a , m , and c are 256 bits. During the calculations, we use 260 bits for the internal signals.

```

`timescale 1ns/1ns // proposed radix-8 implementation for  $c = b * a^{-1} \bmod m$ 
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] ? p : mm : 260'h0; // z2
    wire [259:0] tz4 = t3[0] ? equ ? mm : p : // z4
    wire [259:0] tz8 = t3[0] ? t3[2:1] == p[2:1] ? // z8
    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
        end
    end
endmodule

```

```

end else begin
    if (u == 1) begin
        ready0 <= 1;
        busy    <= 0;
    end else begin
        if (uv[259]) begin
            v <= uv;
            y <= xy;
        end else begin
            u <= uv;
            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'h5076ae274e874c2eb0f7778717c39460236549ddd9fc651e68a0c0e787b4ce8;
        m = 256'hffffffffffffffffffffffffffffffffffffffffffffffffffffffffffc2f;
        #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 is available at 416ns. That is, the calculation takes 208 clock cycles.

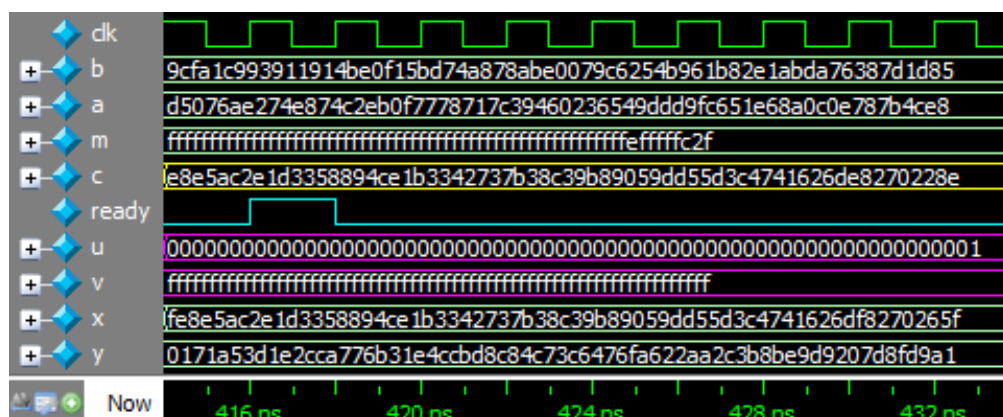


Figure 2. Waveform of modular inversion that calculates $c = ba^{-1} \bmod m$.

We have implemented the modular inversion algorithms on the Altera Cyclone V 5CGXFC9E7F35C8 FPGA chip. Table 5 lists the cost performance of the modular inversion algorithms. The column of Cycles shows the required number of clock cycles when executing the modular inversion algorithm. The column of Freq.(MHz) shows the clock frequency in MHz at which the circuit can work. The column of Latency(μ s) shows the execution time in microseconds calculated by dividing the clock cycles by the clock frequency. The column of ALMs shows the required number of adaptive logic modules. The column of 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 on every clock cycle. The last column shows the AT factor, which is the product of the Latency in milliseconds and the sum of ALMs and Registers.

The row of [1] in the table shows the performance and cost of modular inversion using Fermat’s Little Theorem $r = a^{m-2} \bmod m = a^{-1} \bmod m$. It consists of costly modular multiply and modular squaring, very similar to RSA exponentiation [16]. Its AT factor is much higher than others. The remaining rows show the performance and cost of the EEA-based modular inversion algorithms. The number of registers used by [2], [3], [8], and [10] is larger than 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 achieves an execution time of 3.67 μ s and an AT factor of 8.30, outperforming all other implementations. Figure 3 shows an intuitive view of latency and AT histograms.

Table 5. Comparison of modular inversion algorithms (on Altera Cyclone V FPGA chip).

Algorithm	Cycles	Freq.(MHz)	Latency(μ s)	ALMs	Registers	AT
[1]	66264	57.54	1151.63	2004	2775	5503.66
[2]	534	54.66	9.77	2619	1302	38.31
[3]	535	54.52	9.81	3735	1303	49.42
[4]	358	39.73	9.01	2474	1038	31.64
[5]	1205	64.55	18.67	1596	1043	49.26
[6]	723	72.21	10.01	1968	1042	30.13
[7]	358	63.60	5.63	959	1037	11.24
[8]	423	59.56	7.10	3475	1303	33.92
[9]	356	60.43	5.89	3950	1057	29.50
[10]	423	54.99	7.69	3644	1303	38.05
[11]	334	56.93	5.87	5276	1057	37.15
Ours	208	56.71	3.67	1227	1037	8.30

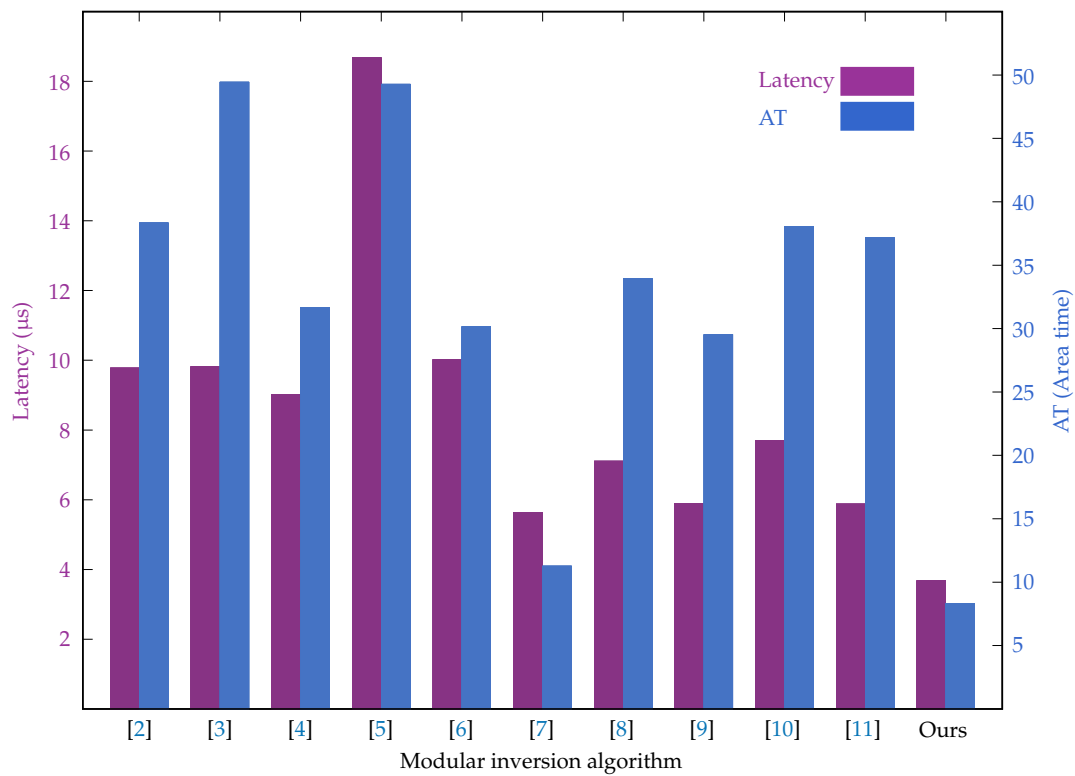


Figure 3. Latency and AT comparison of modular inversion algorithms.

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 $Q = dP$ gets 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 point addition (adding two points) and point doubling, as shown in Algorithm 3.

Algorithm 3 ScaMul (d, P, m, a) (Scalar Point Multiplication in Affine Coordinates).

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** $k \neq 0$ **to**

3 **if** $k \& 1 = 1$

4 $Q = \text{PA}(Q, R, m, a)$

/* $Q = Q + R$ (Algorithm 1) */

5 $R = \text{PD}(R, m, a)$

/* $R = 2R$ (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 6 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 5 steps to obtain $Q = 21P$. We can see that the algorithm is similar to RSA exponentiation [16].

Table 6. 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	$Q = Q + R = 5P + 16P = 21P$	$R = 2R = 32P$

The ECDH algorithm is shown below in Python code that invokes scalar point multiplication four times. The code is hardware oriented. For code integrity, we listed our radix-8 code again but here $+6m$ and $+5m$ are used instead of $-2m$ and $-3m$, respectively.

```

from random import SystemRandom # random number generator
rand = SystemRandom () # strong random number generator
def modadd (a, b, m): # return (a + b) % m; a, b < m
    s = a + b
    if s > m:
        s = s - m
    return s
def modsub (a, b, m): # return (a - b) % m; a, b < m
    s = a - b
    if s < 0:
        s = s + m
    return s
def modmul (a, b, m): # return (a * b) % 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 tx < 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 # 0 + Q
if x2 == -1 and y2 == -1: return P # P + 0
if x1 == x2:
    if modadd (y1, y2, m) == 0: return [-1, -1] # Point 0
    else: return point_doubling (P, m, a) # 2P
# s = ((y1 - y2) / (x1 - x2)) % m
s = modinv (modsub (y1, y2, m), modsub (x1, x2, m), m)
# rx = (s * s - x1 - x2) % m
rx = modsub (modmul (s, s, m), modadd (x1, x2, m), m)
# ry = (s * (x1 - rx) - y1) % 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 0
# s = ((3 * x * x + a) / (2 * y)) % m
s = modinv (modadd(a, modmul(modmul(x, x, m), 3, m), m), modadd(y, y, m), m)
# rx = (s * s - 2 * x) % m
rx = modsub (modmul (s, s, m), modmul (2, x, m), m)
# ry = (s * (x - rx) - y) % 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, b): # scalar point multiplication
if d == 0: return [-1, -1] # Point 0
k = d
Q = [-1, -1] # Point 0
R = P
while k != 0:
    if k & 1:
        Q = point_addition (R, Q, m, a) # PA
        R = point_doubling (R, m, a) # PD
        k >>= 1
    return Q
a = int (0x0000000000000000000000000000000000000000000000000000000000000000)
b = int (0x0000000000000000000000000000000000000000000000000000000000000007)
m = int (0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffc2f)
x = int (0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798)
y = int (0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8)
print ('Secp256k1:')

```

```

print ('a   = 0x{:064x}'.format(a))
print ('b   = 0x{:064x}'.format(b))
print ('m   = 0x{:064x}'.format(m))
print ('x   = 0x{:064x}'.format(x))
print ('y   = 0x{:064x}'.format(y), '\n')
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, b) # Alice's public key
Qb = scalar_point_multiplication ( P, db, m, a, b) # Bob's public key
Qab = scalar_point_multiplication (Qb, da, m, a, b) # Alice calculates shared key
Qba = scalar_point_multiplication (Qa, db, m, a, b) # 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]), '\n')
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 Qab == Qba

```

Based on the above Python code, we implemented ECC using our radix-8 modular inversion algorithm for calculating λ in PA and PD. Figure 4 shows the functional simulation waveform of scalar point multiplication $Q = dP$ with $P = [x, y]$ and $Q = [qx, qy]$. The result c is available at 635362ns. That is, the calculation takes 317681 clock cycles.

The ECC cost performance comparison is given in Table 7 when implementing on the Altera Cyclone V 5CGXFC9E7F35C8 FPGA chip. All ECC implementations use the same circuit except for the modular inversion part. Figure 5 shows the latency and AT histogram. The ECC latency using our proposed radix-8 modular inversion algorithm is 0.02007 second and its AT factor is 401237.93. From the table and histogram, we can see that our ECC implementation achieves a lower latency and a lower AT factor than all others.

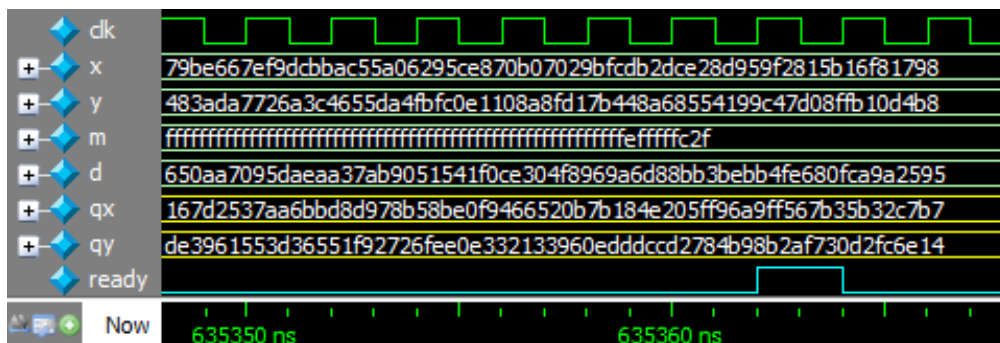


Figure 4. Waveform of scalar point multiplication $Q = dP$ with $P = [x, y]$ and $Q = [qx, qy]$.

Table 7. ECC comparison using modular inversion algorithms (on Altera Cyclone V FPGA chip).

Algorithm	Cycles	Freq.(MHz)	Latency(ms)	ALMs	Registers	AT
[2]	402145	15.94	25.23	15043	8355	590300.42
[3]	402400	15.58	25.83	17585	8355	669977.92
[4]	357262	16.06	22.25	14975	7821	507107.38
[5]	570142	16.07	35.48	13292	7834	749522.08
[6]	455425	16.15	28.20	14211	7831	621577.58
[7]	356878	16.01	22.29	12114	7820	444347.67
[8]	372127	15.98	23.29	17292	8353	597196.30
[9]	352761	15.72	22.44	17841	7860	576737.31
[10]	372127	16.08	23.14	18548	8355	622595.32
[11]	346194	15.88	21.80	20716	7859	622952.99
Ours	317681	15.82	20.08	12157	7824	401237.93

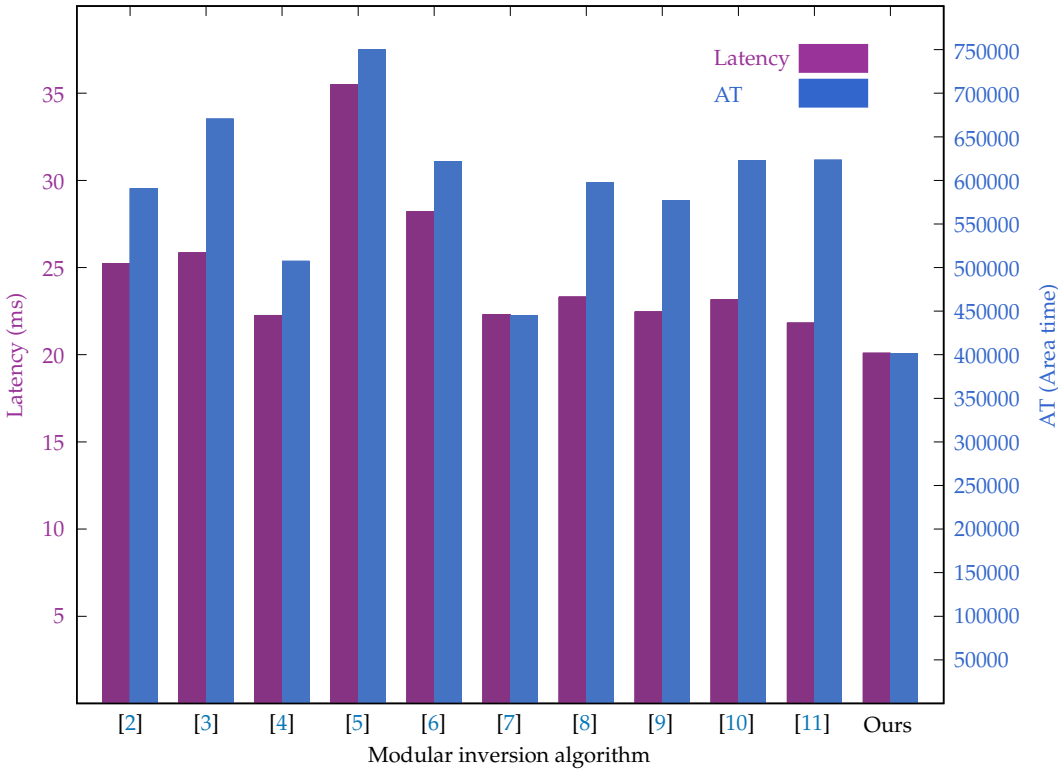


Figure 5. ECC Latency and AT comparison of modular inversion algorithms.

5. Conclusions

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 are area-time efficient with an AT factor of 8.30, which outperforms other algorithms and implementations. We also presented the cost performance of an elliptic curve cryptography implementation using the proposed modular inversion algorithm. Implementation results also show that our algorithm reduces execution time and requires fewer hardware resources than all other investigated algorithms.

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

References

1. Burton, D. *The History of Mathematics / An Introduction (7th ed.)*; McGraw-Hill, 2011. doi:10.1086/355470.
2. Hankerson, D.; Menezes, A.; Vanstone, S. *Guide to Elliptic Curve Cryptography*; Springer-Verlag: New York Inc, 2004. doi:10.1007/b97644.
3. Hossain, M.S.; Kong, Y. High-Performance FPGA Implementation of Modular Inversion over F₂₅₆ for Elliptic Curve Cryptography. 2015 IEEE International Conference on Data Science and Data Intensive Systems, 2015, pp. 169–174. doi:10.1109/DSDIS.2015.47.
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 US: Boston, MA, 2005; pp. 219–229. doi:10.1007/1-4020-3128-9_18.
5. Mrabet, A.; El-Mrabet, N.; Bouallegue, B.; Mesnager, S.; Machhout, M. An efficient and scalable modular inversion/division for public key cryptosystems. 2017 International Conference on Engineering & MIS (ICEMIS), 2017, pp. 1–6. doi:10.1109/ICEMIS.2017.8272995.
6. Chen, C.; Qin, Z. Fast Algorithm and Hardware Architecture for Modular Inversion in GF(p). 2009 Second International Conference on Intelligent Networks and Intelligent Systems, 2009, pp. 43–45. doi:10.1109/ICINIS.2009.20.
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). *Journal of Semiconductor Technology and Science* **2017**, *17*, 425–437. , doi:10.5573/JSTS.2017.17.3.425.
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*, 1–15. <https://www.mdpi.com/2072-666X/14/5/1037>, doi:10.3390/mi14051037.
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. 2019 IEEE 13th International Conference on ASIC (ASICON), 2019, pp. 1–4. doi:10.1109/ASICON47005.2019.8983471.
10. Yan, X.; Li, S. Modified modular inversion algorithm for VLSI implementation. 2007 7th International Conference on ASIC, 2007, pp. 90–93. doi:10.1109/ICASIC.2007.4415574.
11. Dong, X.; Zhang, L.; Gao, X. An Efficient FPGA Implementation of ECC Modular Inversion over F₂₅₆. Proceedings of the 2nd International Conference on Cryptography, Security and Privacy, 2018, pp. 29–33. doi:10.1145/3199478.3199491.
12. Koblitz, N. Elliptic curve cryptosystems. *Mathematics of Computation* **1987**, *48*, 203–209. <https://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866109-5/S0025-5718-1987-0866109-5.pdf>.
13. Miller, V.S. Use of Elliptic Curves in Cryptography. *Advances in Cryptology — CRYPTO '85 Proceedings*; Springer Berlin Heidelberg: Berlin, Heidelberg, 1986; pp. 417–426. https://link.springer.com/content/pdf/10.1007/3-540-39799-X_31.pdf?pdf=inline%20link.
14. Certicom Corp. *Standards for Efficient Cryptography. SEC 2: Recommended Elliptic Curve Domain Parameters*; <http://www.secg.org/sec2-v2.pdf>, 2010.
15. 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, 2018. <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar3.pdf>.
16. Li, Y.; Chu, W. Shift-Sub Modular Multiplication Algorithm and Hardware Implementation for RSA Cryptography. 17th International Conference on Information Assurance and Security, Lecture Notes in Networks and Systems; Springer: Cham, 2021; pp. 541–552. doi:10.1007/978-3-030-96305-7_50.
17. Li, Y. Hardware Implementations of Elliptic Curve Cryptography Using Shift-Sub Based Modular Multiplication Algorithms. *Cryptography* **2023**, *7*, 1–29. , doi:10.3390/cryptography7040057.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.