cryptography

MDPI

Article

Hardware Implementations of Elliptic Curve Cryptography Using Shift-Sub Based Modular Multiplication Algorithms

Yamin Li

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

Abstract: Elliptic curve cryptography (ECC) over prime fields relies on scalar point multiplication realized by point addition and point doubling. Point addition and point doubling operations consist of many modular multiplications of large operands (256 bits for example), especially in projective and Jacobian coordinates which eliminate the modular inversion required in affine coordinates for every point addition or point doubling operation. Accelerating modular multiplication is therefore important for high-performance ECC. This paper presents the hardware implementations of modular multiplication algorithms, including (1) interleaved modular multiplication (IMM), (2) Montgomery modular multiplication (MMM), (3) shift-sub modular multiplication (SSMM), (4) SSMM with advance preparation (SSMMPRE), and (5) SSMM with CSAs and sign detection (SSMMCSA) algorithms, and evaluates their execution time (the number of clock cycles and clock frequency) and required hardware resources (ALMs and registers). Experimental results show that SSMM is 1.80 times faster than IMM, and SSMMCSA is 3.27 times faster than IMM. We also present the ECC hardware implementations based on the Secp256k1 protocol in affine, projective, and Jacobian coordinates using the IMM, SSMM, SSMMPRE, and SSMMCSA algorithms, and investigate their cost and performance. Our ECC implementations can be applied to the design of hardware security module systems.

Keywords: elliptic curve cryptography; affine, projective, and Jacobian coordinates; modular multiplication; hardware security module; Verilog HDL; FPGA; cost/performance evaluation

Citation: Li, Y. Hardware Implementations of Elliptic Curve Cryptography Using Shift-Sub Based Modular Multiplication Algorithms. *Cryptography* **2023**, *7*, 57. https://doi.org/10.3390/cryptography7040057

Academic Editor: Jim Plusquellic

Received: 29 August 2023
Revised: 6 November 2023
Accepted: 7 November 2023
Published: 10 November 2023

1. Introduction

The use of elliptic curves in cryptography was proposed by Neal Koblitz and Victor S. Miller independently in 1985 [1,2]. Elliptic curve cryptography (ECC) over the finite field of a prime number m relies on the fact that scalar point multiplication $Q = dP$ can be computed, but it is almost impossible to compute the multiplicand d given only the original point P and the point of the product Q. Scalar point multiplication can be conducted with point addition (adding two points) and point doubling [3].

In conventional affine coordinates, point addition and point doubling require computing the slope of a line involving division. In both projective and Jacobian coordinates, divisor multiplication is calculated such that division is eliminated during scalar point multiplication. One final division is required to convert a point from projective or Jacobian coordinates to affine coordinates to obtain the shared secret key based on the elliptic curve Diffie–Hellman (ECDH) key exchange. ECDH is a variant of the Diffie–Hellman key agreement protocol using ECC between two parties to establish a shared secret key over an insecure network [4,5]. A modular inversion algorithm calculating $a^{-1} \bmod m$ is given in [3]. Based on this, this paper provides a Verilog HDL implementation to calculate $ba^{-1} \bmod m$. It can be used in affine coordinates to calculate the line slope, or in projective and Jacobian coordinates to convert points to affine coordinates in the final step for obtaining the shared secret key.

Point addition and point doubling operations consist of many modular multiplications, especially in projective and Jacobian coordinates. Accelerating modular multiplication is

therefore important for high-performance ECC. The interleaved modular multiplication (IMM) algorithm [6] eliminates the need for division and has a lower hardware cost. To calculate $p = ab \bmod m$ with $a, b < m$, IMM first calculates $p \leftarrow 2p + b_i a$, where b_i is the ith bit of b. To guarantee $p < m$, we can subtract at most $2m$ from p since $2p + a < 3m$. To carry this out, the traditional IMM algorithm performs the following calculation twice sequentially: if $p \geq m$, $p \leftarrow p - m$. The work in [7] is an FPGA implementation of a processor for elliptic curve point multiplication over prime fields. It uses a modified IMM algorithm with a three-input multiplexer to select one from p, $p - m$, and $p - 2m$. Also, it uses a multiplexer to select one from $2p$ and $2p + a$ for $2p + b_i a$ where b_i is used as the multiplexer selection signal. The design in [8] is similar to [7], but uses two-input "AND" gates to implement $2p + b_i a$ so that it will perform $2p + 0$ if b_i is 0, and $2p + a$ otherwise. The work in [9] is a low hardware consumption elliptic curve cryptographic architecture over prime fields for embedded applications. It performs $p \leftarrow p - p[n+1:n]m$ and uses a four-input multiplexer to select one from 0, $-m$, $-2m$, and $-3m$, for $p[n+1:n] = 0$, 1, 2, and 3, respectively. In fact, a three-input multiplexer is sufficient because $2p + a < 3m$ and hence $p[n+1:n] \neq 3$. The authors in [10] present an ECC architecture that adopts the modular multiplication algorithm proposed in [9]. The design in [11] implements an ECC processor over the NIST [5] prime fields. It uses two subtractors to prepare $p - m$ and $p - 2m$ and then uses a three-input multiplexer to select one from p, $p - m$, and $p - 2m$. It uses two comparison modules to generate the multiplexer selection signals (Figure 3 of paper [11]). This increases hardware costs. In fact, the most significant bit of the 258-bit subtractors can be directly used as the multiplexer selection signals to obtain the final 256-bit product. The authors in [12] describe an implementation of a dual-field ECC processor. The radix-4 IMM algorithm checks two bits of b in each iteration, which reduces the number of iterations by half. Modular multiplications with higher radix require fewer iterations, but the precomputation increases exponentially. The work in [13] is an ECC processor over the NIST P-256 [5] elliptic curve for real-time IoT (Internet of things) applications. It splits the 256-bit inputs into four 64-bit parts and uses the schoolbook-based multiplication algorithm in a pipelined manner to obtain a 512-bit product. Finally, the P-256 fast modular reduction algorithm is used to realize the modular multiplication. Such an implementation can only support the P-256 curve. The Montgomery modular multiplication (MMM) algorithm [14] performs multiplication in the Montgomery domain and is very efficient for modular exponentiation used by RSA cryptography. The modular exponentiation can be calculated by repeatedly calling MMM. However, transformations to the Montgomery domain are required before calculations, and a transformation back to the regular domain is also required to obtain the final result. Domain transformation requires the value $q = R^2 \bmod m$, which is calculated by an expensive modular operation. Using a precomputation of q for fixed R and m speeds up the calculations but reduces the flexibility of using different moduli m. The shift-sub modular multiplication (SSMM) algorithm uses shifts and subtractions to perform modular multiplication. The SSMM algorithm and its use in RSA cryptography are described in [15,16]. An algorithm using CSA (Carry save adder) [17] is proposed but it requires either a modular computation after the iterations or a precomputed look-up table for fixed multiplier and moduli.

This paper focuses on radix-2 algorithms and proposes two enhanced versions of SSMM: SSMM with advance preparation (SSMMPRE) and SSMM with CSAs and sign detection (SSMMCSA) algorithms. We also present the ECC implementations based on Secp256k1 [4] protocol which is used in the Ethereum blockchain. The specific contributions of this paper are summarized as follows: (1) Based on SSMM, we propose SSMMPRE (SSMM with advance preparation) and SSMMCSA (SSMM with CSAs and sign detection) algorithms that have lower latency than SSMM. (2) The hardware implementations of IMM, MMM, SSMM, SSMMPRE, and SSMMCSA algorithms are presented and their cost (required adaptive logic modules (ALMs) and registers) and performance (the number of clock cycles and clock frequency) are evaluated. (3) The hardware implementations of ECC in affine, projective, and Jacobian coordinates using the IMM, SSMM, SSMMPRE, and

SSMMCSA algorithms are presented and their cost and performance are evaluated and compared with those proposed in [7–10]. (4) Some important Verilog HDL source codes and their simulation waveform are included in Appendices A and B. The experimental results show that SSMM is 1.80 times faster than IMM and SSMMCSA is 3.27 times faster than IMM, and our ECC implementations perform better than those proposed in [7–10]. Compared to RSA cryptography, ECC provides stronger encryption with shorter key lengths [18]. The ECC implementations described in this paper can support other elliptic curves, such as the NIST P-256 (Secp256r1) [5] curve.

The rest of the paper is organized as follows. Section 2 introduces the background of ECC, including the point addition, point doubling, scalar point multiplication, modular inversion, ECDH key agreement protocol, and affine, projective, and Jacobian coordinates. Section 3 describes the modular multiplication algorithms, including the IMM, MMM, SSMM, SSMMPRE, and SSMMCSA algorithms. Section 4 presents hardware implementations of modular multiplications and ECC and evaluates their cost and performance. Comparisons with [7–10] are also given in this section. And Section 5 concludes the paper. Verilog HDL codes for SSMM and modular inversion are listed in Appendices A and B.

2. Elliptic Curve Cryptography Algorithms

This section describes ECC algorithms and ECDH that underlie this work. A top view of the relationship between these algorithms is shown in Figure 1.

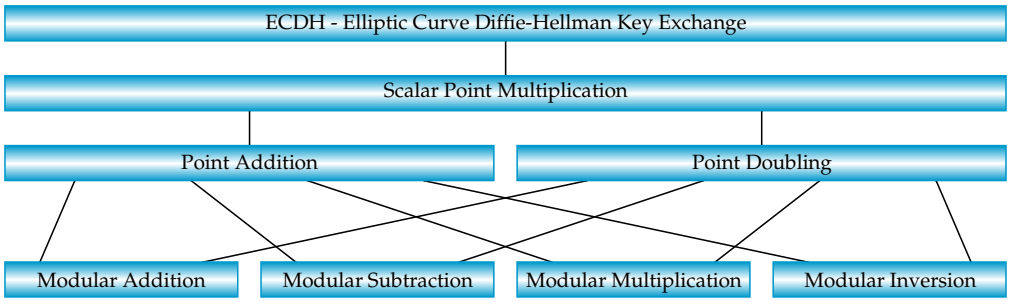

Figure 1. ECDH and ECC algorithms. The ECDH key exchange invokes a scalar point multiplication that uses two computations—point addition and point doubling. Four primitive modular calculations (addition, subtraction, multiplication, and inversion) are used for these two computations.

Modular multiplication and modular inversion are performed with iterations which will be described later. Modular addition and modular subtraction are calculated in one clock cycle. The Verilog HDL codes for modular addition and modular subtraction used in our ECC implementations are listed below. The most significant bits of 258-bit subtractors `s_m[257]` and `sum[257]` are used as the selection signals of the multiplexers.

```
module modadd (a, b, m, s); // s = (a + b) mod m
    input  [255:0] a, b, m;
    output [255:0] s;
    wire   [257:0] sum = {2'b00,a} + {2'b00,b};
    wire   [257:0] s_m = sum - {2'b00,m};
    assign         s   = s_m[257] ? sum[255:0] : s_m[255:0];
endmodule

module modsub (a, b, m, s); // s = (a - b) mod m
    input  [255:0] a, b, m;
    output [255:0] s;
    wire   [257:0] sum = {2'b00,a} - {2'b00,b};
    wire   [257:0] s_m = sum + {2'b00,m};
    assign         s   = sum[257] ? s_m[255:0] : sum[255:0];
endmodule
```

The interesting computations in ECC are point addition and point doubling. We first introduce these two computations in an elliptic curve over the real numbers. In the real number field, an elliptic curve can be defined in Weierstrass form as

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

Note that this curve is symmetrical about the x-axis. If point $P = [x_p, y_p]$ is on an elliptic curve, then $-P = [x_p, -y_p]$ is also on the same elliptic curve.

2.1. ECC Point Addition and Doubling in Affine Coordinates

Affine coordinates use two coordinates $[x, y]$ to represent an elliptic curve point, as shown by, for example, $P = [x_p, y_p]$ and $-P = [x_p, -y_p]$. We will see that the ECC point addition and point doubling in affine coordinates require expensive divisions.

2.1.1. ECC Point Addition in Affine Coordinates

Figure 2 shows the point addition $R = P + Q$ on an elliptic curve $y^2 = x^3 + ax + b$. Given two distinct points $P = [x_p, y_p]$ and $Q = [x_q, y_q]$ on the curve, if the line L_1 through P and Q intersects the curve in $S = [x_s, y_s]$, then $R = [x_r, y_r] = P + Q$ is defined as $x_r = x_s$ and $y_r = -y_s$. Because $x_r = x_s$, the line L_2 through S and R is a vertical line.

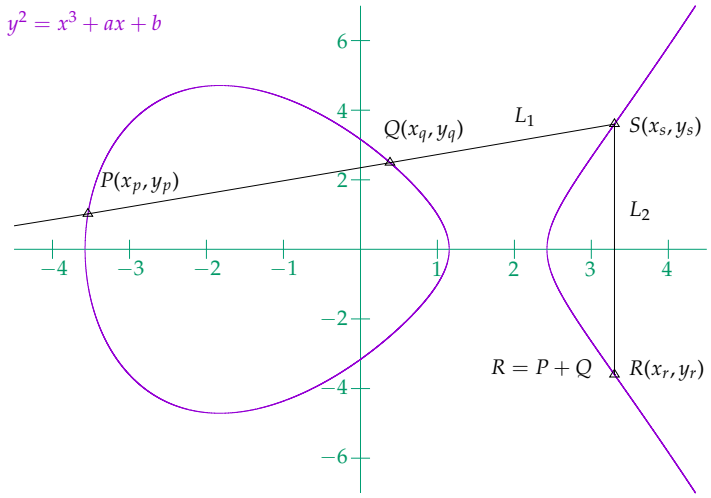

Figure 2. Point addition $R = P + Q$ on an elliptic curve $y^2 = x^3 + ax + b$ in the real number field.

Below we show how to obtain formulas to calculate x_r and y_r based on x_p, y_p, x_q, and y_q for $y^2 = x^3 + ax + b$. The formula of the line L_1 through P and Q is

$$y = \lambda(x - x_p) + y_p \tag{2}$$

where λ is the slope of the line L_1. Squaring both the left side and right side of Equation (2), we obtain $y^2 = (\lambda(x - x_p) + y_p)^2$. Then, replacing y^2 of Equation (1) with $(\lambda(x - x_p) + y_p)^2$, we have

$$x^3 - \lambda^2 x^2 + (a - 2y_p\lambda + 2x_p\lambda^2)x + (b - (y_p - \lambda x_p)^2) = 0 \tag{3}$$

Because P, Q, and S are three points on the curve, meaning that x_p, x_q, and x_s are three roots of Equation (3), based on Vieta's formulas, we have

$$(x - x_p)(x - x_q)(x - x_s) = 0 \tag{4}$$

Expanding Equation (4) gives:

$$x^3 - (x_p + x_q + x_s)x^2 + (x_p x_q + x_q x_s + x_s x_p)x - x_p x_q x_s = 0 \tag{5}$$

Then from Equations (3) and (5) we have $x_p + x_q + x_s = \lambda^2$. That is, $x_s = \lambda^2 - x_p - x_q$ and $y_s = \lambda(x_s - x_p) + y_p$. Considering the L_1 line slope $\lambda = (y_q - y_p)/(x_q - x_p)$, $x_r = x_s$, and $y_r = -y_s$, we summarize the formulas for point addition $R = P + Q = (x_r, y_r)$ on elliptic curve $y^2 = x^3 + ax + b$ as follows.

$$\lambda = \frac{y_q - y_p}{x_q - x_p} \tag{6}$$

$$x_r = \lambda^2 - x_p - x_q \tag{7}$$

$$y_r = \lambda(x_p - x_r) - y_p \tag{8}$$

For $Q = -P$, the line through P and $-P$ does not intersect the elliptic curve at the third point. For this reason, the point O at infinity is included in the group of elliptic curves and defined as $P + (-P) = O$. By this definition, $P + O = P$.

In practice, a group of elliptic curves over a finite field of F_m or F_{2^n} is used, where F_m contains numbers from 0 to $m - 1$ and F_{2^n} uses n-bit binary numbers. In the case of F_m, the results of all the above calculations are modularized by m, where m is usually a prime number. For example, Secp256k1 [4] 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
```

By considering $P + O = P$, we give the point addition $R = P + Q$ algorithm over the finite field of F_m in Algorithm 1. In our implementation, O is denoted as $[-1, -1]$. In the case of $Q = P$, we perform the point doubling $R = 2P$ (line 6 in the algorithm).

Algorithm 1 PAA (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 $x_p = P_x, y_p = P_y, x_q = Q_x, y_q = Q_y, O = [-1, -1]$
2 **if** $P = O$ **return** Q /* $O + Q = Q$ */
3 **if** $Q = O$ **return** P /* $P + O = P$ */
4 **if** $x_p = x_q$
5 **if** $(y_p + y_q) \bmod m = 0$ **return** O /* $P + (-P) = O$ */
6 **else return** PDA (P, p, 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 = 0xe493dbf1c10d80f3581e4904930b1404cc6c13900ee0758474fa94abe8c4cd13
Py = 0x51ed993ea0d455b75642e2098ea51448d967ae33bfbdfe40cfe97bdc47739922
Qx = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
Qy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
Rx = 0x2f8bde4d1a07209355b4a7250a5c5128e88b84bddc619ab7cba8d569b240efe4
Ry = 0xd8ac222636e5e3d6d4dba9dda6c9c426f788271bab0d6840dca87d3aa6ac62d6
```

2.1.2. ECC Point Doubling in Affine Coordinates

Figure 3 shows the point doubling $R = 2P$ on an elliptic curve $y^2 = x^3 + ax + b$. Compared to the point addition shown in Figure 2, here we have $Q = P$ and $R = P + Q = 2P$. Given a point $P = [x_p, y_p]$ on the curve, if the tangent line L_1 through P intersects the curve in $S = [x_s, y_s]$, then $R = [x_r, y_r] = 2P$ is defined as $x_r = x_s$ and $y_r = -y_s$.

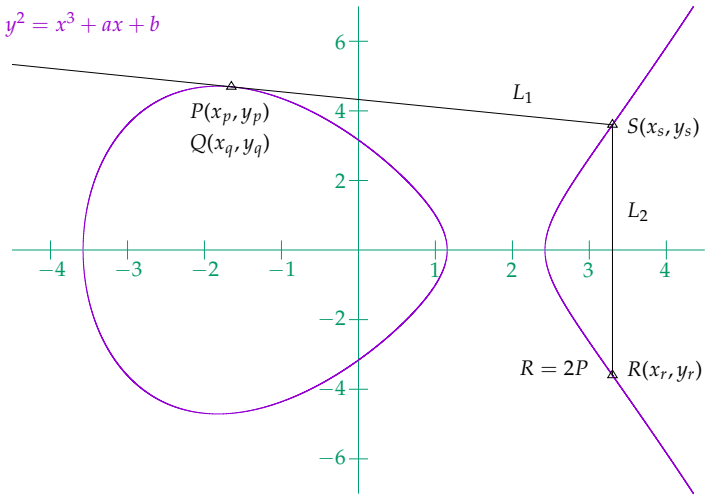

Figure 3. Point doubling $R = 2P$ on an elliptic curve $y^2 = x^3 + ax + b$ in the real number field.

Below we show how to obtain formulas to calculate x_r and y_r based on x_p, y_p, and a for $y^2 = x^3 + ax + b$. The formula of the line L_1 through P is

$$y = \lambda(x - x_p) + y_p \tag{9}$$

where λ is the slope of the line L_1. Squaring both the left side and right side of Equation (9), we obtain $y^2 = (\lambda(x - x_p) + y_p)^2$. Then, replacing y^2 of Equation (1) with $(\lambda(x - x_p) + y_p)^2$, we have

$$x^3 - \lambda^2 x^2 + (a - 2y_p\lambda + 2x_p\lambda^2)x + (b - (y_p - \lambda x_p)^2) = 0 \tag{10}$$

Because P, Q $(= P)$, and S are three points on the curve, meaning that x_p, x_q, and x_s are three roots of Equation (10), based on Vieta's formulas, we have

$$(x - x_p)^2(x - x_s) = 0 \tag{11}$$

Expanding Equation (11) gives:

$$x^3 - (2x_p + x_s)x^2 + (x_p^2 + 2x_p x_s)x - x_p^2 x_s = 0 \tag{12}$$

Then from Equations (10) and (12) we have $2x_p + x_s = \lambda^2$. That is, $x_s = \lambda^2 - 2x_p$ and $y_s = \lambda(x_s - x_p) + y_p$. The slope of the tangent line L_1 of $y^2 = x^3 + ax + b$ at P can be obtained as follows.

$$\frac{d}{dx}(y^2) = \frac{d}{dx}(x^3 + ax + b) \tag{13}$$

$$2y\frac{dy}{dx} = 3x^2 + a \tag{14}$$

$$\frac{dy}{dx} = (3x^2 + a)/(2y) \tag{15}$$

$$\lambda = (3x_p^2 + a)/(2y_p) \text{ at } P \tag{16}$$

Considering $x_r = x_s$ and $y_r = -y_s$, we summarize the formulas for point doubling $R = 2P$ on elliptic curve $y^2 = x^3 + ax + b$ as follows.

$$\lambda = \frac{3x_p^2 + a}{2y_p} \tag{17}$$

$$x_r = \lambda^2 - 2x_p \tag{18}$$

$$y_r = \lambda(x_p - x_r) - y_p \tag{19}$$

We give the point doubling $R = 2P$ algorithm over the finite field of F_m in Algorithm 2. For $P_y = 0$, the tangent at P is vertical and does not intersect the elliptic curve at any other point. By definition, $2P = O$ for such a point P (line 2 in the algorithm).

Algorithm 2 PDA (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 $x_p = P_x, y_p = P_y, O = [-1, -1]$
2 **if** $y_p = 0$ **return** 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 = 0xb91dc87409c8a6b81e8d1be7f5fc86015cfa42f717d31a27d466bd042e29828d
Py = 0xc35b462fb20bec262308f9d785877752e63d5a68e563e898b4f82f47594680fc
Rx = 0x2d4fca9e0dff8dec3476a677d555896a0980ebccc6bc595a23675496dcc33bb5
Ry = 0xcce413eee9496094256e446b22fd234c03d9258330d77fc8b0d318a6aedba8cb
```

2.2. ECC Point Addition and Doubling in Projective Coordinates

Point addition and point doubling in affine coordinates require modular inversion (division) to calculate the line slope λ. We can eliminate the expensive modular inversion during calculations by using projective coordinates. In projective coordinates, a point is defined as $P = [X, Y, Z]$. Initially, we can convert a point $[X, Y]$ in affine coordinates to a point in projective coordinates by $P = [X, Y, 1]$. Then, we calculate $R = [X_r, Y_r, Z_r]$ in projective coordinates using formulas that do not contain division. At the very final step, we can obtain the point $[x_r, y_r]$ in affine coordinates with the transformation of $x_r = X_r/Z_r$ and $y_r = Y_r/Z_r$, which requires divisions.

The formulas for ECC point addition and doubling in projective coordinates can be derived based on the formulas in affine coordinates. A point P in projective coordinates is represented by the triple $P = [X, Y, Z]$, corresponding to the point $[x_p, y_p] = [X/Z, Y/Z]$ in affine coordinates. That is, $x_p = X/Z$ and $y_p = Y/Z$. We derive the point doubling formulas for $R = [X_r, Y_r, Z_r] = 2P$ in projective coordinates as follows.

From Equation (17), we have

$$\lambda = \frac{3x_p^2 + a}{2y_p} = \frac{3(X/Z)^2 + a}{2(Y/Z)} = \frac{3X^2/Z^2 + a}{2Y/Z} = \frac{3X^2 + aZ^2}{2YZ}$$

From Equation (18), we have

$$x_r = \lambda^2 - 2x_p = (\frac{3X^2 + aZ^2}{2YZ})^2 - 2X/Z = \frac{(3X^2 + aZ^2)^2}{(2YZ)^2} - \frac{8XY^2Z}{(2YZ)^2}$$

From Equation (19), we have

$$y_r = \lambda(x_p - x_r) - y_p = \frac{3X^2 + aZ^2}{2YZ}(X/Z - \frac{(3X^2 + aZ^2)^2 - 8XY^2Z}{(2YZ)^2}) - Y/Z$$
$$= \frac{(3X^2 + aZ^2)(4XY^2Z - ((3X^2 + aZ^2)^2 - 8XY^2Z))}{(2YZ)^3} - \frac{8Y^4Z^2}{(2YZ)^3}$$

Let $Z_r = (2YZ)^3$.

Because $x_r = X_r/Z_r = X_r/(2YZ)^3$, i.e., $X_r = (2YZ)^3 x_r$, then

$$X_r = 2YZ((3X^2 + aZ^2)^2 - 8XY^2Z)$$

Because $y_r = Y_r/Z_r = Y_r/(2YZ)^3$, i.e., $Y_r = (2YZ)^3 y_r$, then

$$Y_r = (3X^2 + aZ^2)(4XY^2Z - ((3X^2 + aZ^2)^2 - 8XY^2Z)) - 8Y^4Z^2$$

Given $P = [X, Y, Z]$, the formulas for point doubling $R = 2P$ in projective coordinates are summarized below.

$$X_r = 2YZ((3X^2 + aZ^2)^2 - 8XY^2Z) \tag{20}$$

$$Y_r = (3X^2 + aZ^2)(4XY^2Z - ((3X^2 + aZ^2)^2 - 8XY^2Z)) - 8Y^4Z^2 \tag{21}$$

$$Z_r = (2YZ)^3 \tag{22}$$

We can use a similar method to derive the formulas for point addition in projective coordinates. The derivation is omitted here but the calculations are shown in the following algorithm. It can be seen that the calculations require many more multiplications. Algorithm 3 gives the algorithm for point addition in projective coordinates. And Algorithm 4 gives the algorithm for point doubling in projective coordinates.

Algorithm 3 PAP (P, Q, m, a) (point addition in projective coordinates).

inputs: Points $P = [P_x, P_y, P_z]$ and $Q = [Q_x, Q_y, Q_z]$; m and a in $y^2 = x^3 + ax + b \bmod m$
output: $R = P + Q = [R_x, R_y, R_z] = [x_r, y_r, z_r]$
begin

$u = P_x, v = P_y, w = P_z, x = Q_x, y = Q_y, z = Q_z, O = [-1, -1, -1]$
if $P = O$ **return** Q /* $O + Q = Q$ */
if $Q = O$ **return** P /* $P + O = P$ */
if $u = x$
  if $(v + y) \bmod m = 0$ **return** O /* $P + (-P) = O$ */
  else return PDP (P, p, a) /* $P + P = 2P$ */
$s = vz - wy, t = uz - wx, h = uz + wx$ /* level 1 calculations */
$k = s^2wz - t^2h, n = t^3z$ /* level 2 calculations */
$x_r = tk \bmod m$ /* level 3 calculations */
$y_r = (s(uzt^2 - k) - vn) \bmod m$ /* level 3 calculations */
$z_r = wn \bmod m$ /* level 3 calculations */
return $[x_r, y_r, z_r]$ /* $R = P + Q$ */

end

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

```
Px = 0x61bac660b055382e5906bd6e56e316542194b799b7bcf5ad05ee2171fd81735a
Py = 0xbe44ac0a2b712ccb6bb3ea933e4db0a4213c139078aef594cf8c2c5c2924d54d
Pz = 0x2b6da6fb02877584dc4d5111c88783772d7be5ac2866cce3707d53913384bf49
Qx = 0x6789c1137724f1f3f585337a1814eebc23ea329a0390fd9b1b9ece7af3e71ce1
Qy = 0xc9563c5035ccafec8673f56185141f720073ab3063bb417bf0e70e9d9128c232
Qz = 0x58990cd022b711912676c0451bdab6be04a06c1871b0139214bdbe81fd965555
Rx = 0xf4e9cb9ba9c18876b7b0ad000ce921b35e23139456f4f6c3f70e2fea149500a0
Ry = 0xc06176a9221b6d8b49a22130fb934b21358a1775df68d93ec308aca3ece072b5
Rz = 0x878fb153f0690416ba0ee136ec663debf8472f3ee92d350f9b3a42b4fd53fb27
```

Algorithm 4 PDP (P, m, a) (point doubling in projective coordinates).

inputs: Point $P = [P_x, P_y, P_z]$; m and a in $y^2 = x^3 + ax + b \bmod m$
output: $R = 2P = [R_x, R_y, R_z] = [x_r, y_r, z_r]$
begin
1 $x = P_x, y = P_y, z = P_z, O = [-1, -1, -1]$
2 **if** $y_p = 0$ **return** O /* vertical tangent */
3 $s = 3x^2 + az^2, t = 4y^2z$ /* level 1 calculations */
4 $h = 2yzt, k = s^2 - 2xt$ /* level 2 calculations */
5 $x_r = 2yzk \bmod m$ /* level 3 calculations */
6 $y_r = (s(xt - k) - yh) \bmod m$ /* level 3 calculations */
7 $z_r = zh \bmod m$ /* level 3 calculations */
8 **return** $[x_r, y_r, z_r]$ /* $R = 2P$ */
end

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

```
Px = 0x24fd537e9a5125438a02848f6b74725f678723f5c1450b8fb82a68f0c88c9764
Py = 0xe42e83a1d3d7c2241535b5c0ba5f2462c24bd87aaf9f15b05f3775d168b9bf6c
Pz = 0xe00794d20b32e0e94472c36b89cf5e5d6ec769b53dd6c1422e9467090c272305
Rx = 0x24d00c48ac8bbe61ceb0ac5daf5defd913af9220a07650642a3a41cad9030ee6
Ry = 0x75d429714ea6ce1ab3811d9adc16961a219e2812210fa8465042c18ecd5a0de6
Rz = 0x644a5a2964435364b74d7fa79fe0f06a5b1d2782e7f7b8d1e835db6d6b8786bc
```

2.3. ECC Point Addition and Doubling in Jacobian Coordinates

A point P in Jacobian coordinates is represented by the triple $P = [X, Y, Z]$, corresponding to the point $[x_p, y_p] = [X/Z^2, Y/Z^3]$ [2] (p. 424) in affine coordinates. That is, $x_p = X/Z^2$ and $y_p = Y/Z^3$. We derive the point doubling formulas for $R = [X_r, Y_r, Z_r] = 2P$ in Jacobian coordinates as follows.

From Equation (17), we have

$$\lambda = \frac{3x_p^2 + a}{2y_p} = \frac{3(X/Z^2)^2 + a}{2(Y/Z^3)} = \frac{3X^2/Z^4 + a}{2Y/Z^3} = \frac{3X^2 + aZ^4}{2YZ}$$

From Equation (18), we have

$$x_r = \lambda^2 - 2x_p = (\frac{3X^2 + aZ^4}{2YZ})^2 - 2X/Z^2 = \frac{(3X^2 + aZ^4)^2}{(2YZ)^2} - \frac{8XY^2}{(2YZ)^2}$$

From Equation (19), we have

$$y_r = \lambda(x_p - x_r) - y_p = \frac{3X^2 + aZ^4}{2YZ}(X/Z^2 - X_r/(2YZ)^2) - Y/Z^3$$

$$= \frac{(3X^2 + aZ^4)(4XY^2 - X_r)}{(2YZ)^3} - \frac{8Y^4}{(2YZ)^3}$$

Let $Z_r = 2YZ$.
Because $x_r = X_r/Z_r^2 = X_r/(2YZ)^2$, we have

$$X_r = (3X^2 + aZ^4)^2 - 8XY^2$$

Because $y_r = Y_r/Z_r^3 = Y_r/(2YZ)^3$, we have

$$Y_r = (3X^2 + aZ^4)(4XY^2 - X_r) - 8Y^4$$

Given $P = [X, Y, Z]$, the formulas for point doubling $R = 2P$ in Jacobian coordinates are summarized below.

$$X_r = (3X^2 + aZ^4)^2 - 8XY^2 \tag{23}$$

$$Y_r = (3X^2 + aZ^4)(4XY^2 - X_r) - 8Y^4 \tag{24}$$

$$Z_r = 2YZ \tag{25}$$

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

```
Px = 0xe43306185ef298127aef469d577aed78acafaddfc28ad0857491c38ffbedc475
Py = 0x4d83871239769596f65c180546c170a28cffca37bf6393025c457f406f54c517
Pz = 0xdafa620812722dceda7d93a91158dadbe11fee894e71eafa054d5f5fd274377e
Rx = 0x7d7cd6974d7e127a5fdf3f3c9c9eb5dcd9c15e033794466de63bcf2b9548ff85
Ry = 0x8f967b514296945a6dd052bca59ec1418a35cde3c6dd7b269d2e71daa80f851e
Rz = 0xcf89daf8b6c736cc851882b7e85c8ea8f703a9323a3d627909582b7904766035
```

Based on Equations (6)–(8), and $x = X/Z^2$ and $y = Y/Z^3$, the formulas for point addition in Jacobian coordinates can be derived which are omitted here. An example of point addition $R = P + Q$ on the Secp256k1 curve is shown below where $[P_x, P_y, P_z] = P$, $[Q_x, Q_y, Q_z] = Q$, and $[R_x, R_y, R_z] = R$ in Jacobian coordinates.

```
Px = 0xb7bae589ec8a8c722c1ffb2c37fd4bbeda59074675c3eb50f1673ed46bbedfbe
Py = 0x81dee3398bdd718591c10762f61a0e41c4d609dffddcbeeb3894b8c4ce75e027
Pz = 0x51ec57b21350ad3d3466be5a7d28742279fbac1146fb4143767ee368a7dc741e
Qx = 0x0aa20a04dba4788e9b99e10f2e9f4d43b7f53916a5cacf2050dc70bc34c18d21
Qy = 0x60f318d01180b303f4b20a49c2b7e2b498405f88bd423a9a7cb92bab5f1b6abf
Qz = 0x3e1dcb6efa88b113f40b5858ea8c3cb5ddae2277c4683af9487e27023cba690d
Rx = 0x348248c47ad5d3186bd807c382659263840ba7ea13e61128d24337db9b0e5278
Ry = 0xb38573698dd6fef9ec93a9d68ae0a997191b678474cc00a13961defa3ed763e9
Rz = 0x76da2008046aae9901e6b96a7b54c42fd480de5cbc8cef6c0d0a9b3d7086f0f4
```

The point $[x_r, y_r]$ in affine coordinates can be obtained from Jacobian coordinates by $x_r = X_r/Z_r^2$ and $y_r = Y_r/Z_r^3$. This is only performed one time at the final step. It can be conducted with the modular inversion which we will introduce next.

2.4. Modular Inversion

In affine coordinates, the point addition and point doubling algorithms must compute the slope λ of a line. Projective and Jacobian coordinates require the point to be transformed into affine coordinates at the final step to obtain the shared secret key. These calculations or transformations require division and modulo operations.

Generally, the modular inversion calculates $c = ba^{-1} \bmod m$, where m is an n-bit odd number and $\{a, b\} < m$. An example of modular inversion is shown below.

```
b = 0x9cfa1c993911914be0f15bd74a878abe0079c6254b961b82e1abda76387d1d85
a = 0xd5076ae274e874c2eb0f7778717c39460236549ddd9fc651e68a0c0e787b4ce8
m = 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f
c = 0xe8e5ac2e1d3358894ce1b3342737b38c39b89059dd55d3c4741626de8270228e
```

Algorithm 2.22 in [3] gives an algorithm to calculate $a^{-1} \bmod m$ using the extended Euclidean algorithm. Based on this, we give a Verilog HDL implementation of the modular inversion that calculates $c = ba^{-1} \bmod m$ in Appendix B. Figure A2 shows the simulation waveform generated with ModelSim.

In affine coordinates, every point addition or point doubling invokes modular inversion to calculate the line slope λ. In projective or Jacobian coordinates, this modular inversion can be removed during point addition or point doubling calculations, but a final division is required to transform a point from projective or Jacobian to affine coordinates to obtain the shared secret key. This division can be achieved using modular inversion.

2.5. Scalar Point Multiplication

Scalar point multiplication performs $Q = dP$ where P and Q are elliptic curve points and $d = \langle d_{n-1} \cdots d_1 d_0 \rangle$ is an n-bit scalar. Scalar point multiplication can be conducted with the "double-and-add" method [3]. Algorithm 5 formally gives the algorithm for scalar point multiplication. The algorithm invokes point addition and point doubling. Point doubling can be calculated simultaneously with point addition.

Algorithm 5 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 $Q = O, R = P, k = d$ /* $Q = O$ and $R = P$ */
2 **while** $k \neq 0$ **to**
3 **if** $k_0 = 1$
4 $Q = Q + R$ /* point addition */
5 $R = 2R$ /* point doubling */
6 $k = k \gg 1$
7 **endwhile**
8 **return** Q /* $Q = dP$ */
end

Below we give an example to show the calculation steps of the scalar point multiplication. For a 5-bit $d = 11011_2 = 27$, we calculate $Q = dP$ in 5 steps to obtain $Q = 27P$.

	Weight	**Point Addition**	**Point Doubling**
Initial		$Q = O$	$R = P$
$d_0 = 1$	1	$Q = Q + R = O + P = P$	$R = 2R = 2P$
$d_1 = 1$	2	$Q = Q + R = O + 2P = 3P$	$R = 2R = 4P$
$d_2 = 0$	4		$R = 2R = 8P$
$d_3 = 1$	8	$Q = Q + R = 3P + 8P = 11P$	$R = 2R = 16P$
$d_4 = 1$	16	$Q = Q + R = 11P + 16P = 27P$	$R = 2R = 32P$

2.6. Elliptic Curve Diffie–Hellman Key Exchange

The elliptic curve Diffie–Hellman (ECDH) algorithm is a variant of the Diffie–Hellman key agreement protocol using ECC between two parties to establish a shared secret key over an insecure network [4,5]. Then, this shared secret key can be used by the two parties to encrypt and decrypt subsequent communications using fast symmetric-key cryptography over the insecure network. The ECDH key exchange 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 Calculate $Q_a = d_a P$ (Algorithm 5) Expose Q_a	Generate a secret d_b Calculate $Q_b = d_b P$ (Algorithm 5) Expose Q_b
Get Q_b from Bob Calculate $Q_{ab} = d_a Q_b$ (Algorithm 5)	Get Q_a from Alice Calculate $Q_{ba} = d_b Q_a$ (Algorithm 5)
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 two parties have the same shared secret key (Qabx = Qbax).

Alice generates and exposes $Q_a = d_a P$:

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

Bob generates 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 a same secret key ($Qabx = Qbax$). They can use a symmetric-key cryptography for the subsequent communications.

3. Modular Multiplication Algorithms

Point addition and point doubling use many modular multiplications. This section describes interleaved modular multiplication (IMM), Montgomery modular multiplication (MMM), and shift-sub modular multiplication (SSMM) algorithms, and proposes shift-sub modular multiplication with advance preparation (SSMMPRE) and shift-sub modular multiplication with CSAs and sign detection (SSMMCSA) algorithms.

3.1. Interleaved Modular Multiplication Algorithm

The IMM algorithm [6] is formally given in Algorithm 6. It computes $p = ab \bmod m$ where $a, b < m < 2^n$, and m is an n-bit odd number. That is, the $(n-1)$th bit and 0th bit of n-bit m are 1. IMM begins with checking the $(n-1)$th bit of multiplier b. Therefore, in each iteration, the product p is shifted to the left by one bit (line 3). Because $a, b < m$, $2p + a < 3m$ (lines 3 and 4). Therefore, it is enough to subtract $2m$ from $2p + a$ (lines 5 and 6), ensuring $p < m$.

Algorithm 6 IMM (a, b, m) (interleaved modular multiplication).

inputs: $a = \sum_{i=0}^{n-1} a_i 2^i$, $b = \sum_{i=0}^{n-1} b_i 2^i$, $a, b < m < 2^n$, m: n-bit odd number
output: $p = ab \bmod m$
begin
1 $p \leftarrow 0$ /* product */
2 **for** $i = n-1$ **downto** 0
3 $p \leftarrow p \ll 1$ /* $p = 2p$ */
4 $p \leftarrow p + b_i a$ /* add multiplicand a to p if $b_i = 1$ */
5 **if** $p \geq m$, $p \leftarrow p - m$ /* subtract m from p */
6 **if** $p \geq m$, $p \leftarrow p - m$ /* subtract m from p */
7 **return** p
end

Figure 4 shows a possible IMM hardware implementation of Algorithm 6. Red rectangles are registers, others are combinational circuits. Clearly, the critical path is the right part that computes the new p in each iteration. It consists of three carry propagate adders (CPAs) and three multiplexers. The most significant bit of the adder output (sign) can be used as the select signal of multiplexers. Note that $p \ll 1$ can be realized by wiring.

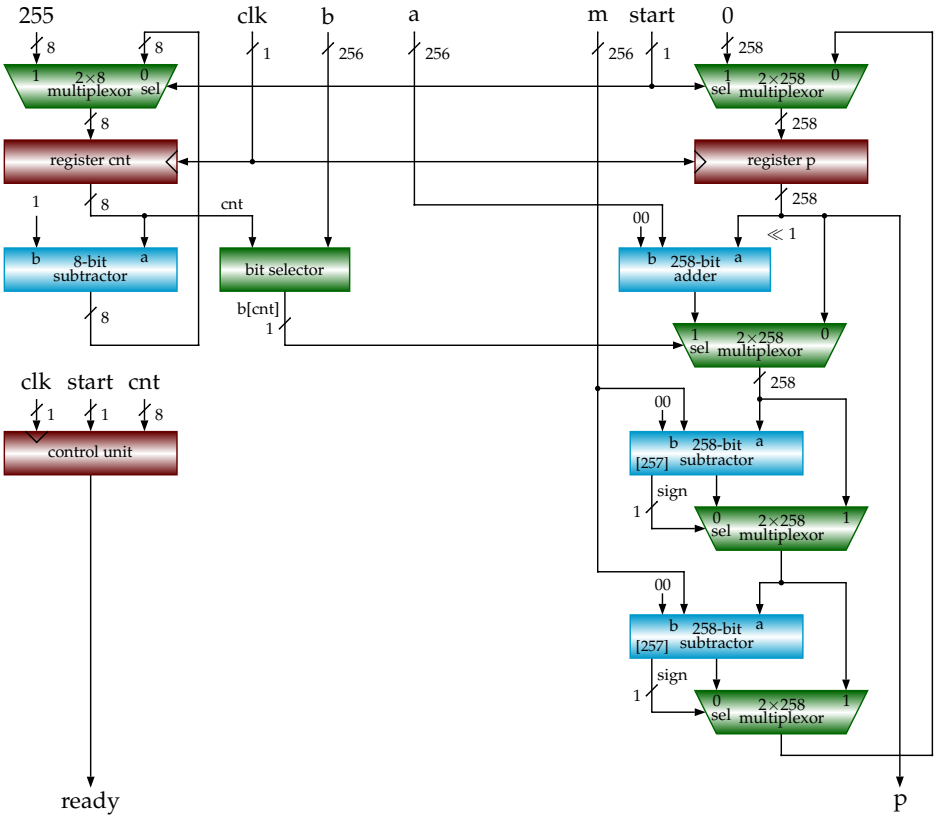

Figure 4. Block diagram of interleaved modular multiplication (IMM). It implements Algorithm 6.

3.2. Montgomery Modular Multiplication Algorithm

The MMM algorithm [14] performs modular multiplication in the Montgomery domain. It is very efficient for modular exponentiation used by RSA cryptography. MMM calculates $p = abR^{-1} \bmod m$, where $R = 2^n$, $a, b < m < R$, and m is an n-bit odd number with $m_{n-1} = m_0 = 1$. It performs reduction R^{-1} during the multiplication ab because a and b are represented in the Montgomery domain as follows.

$$a = a'R \bmod m \tag{26}$$

$$b = b'R \bmod m \tag{27}$$

where a' and b' are the operands in the conventional domain. Such a reduction ensures that the product p is an operand still in Montgomery domain:

$$p = abR^{-1} \bmod m = a'Rb'RR^{-1} \bmod m = a'b'R^2R^{-1} \bmod m = a'b'R \bmod m \tag{28}$$

MMM is widely used in RSA cryptography where the modular exponentiation is realized with the repeated modular multiplications. A bit-level MMM algorithm is formally given in Algorithm 7. For the reduction, we perform

$$R^{-1} = \frac{1}{2^n} = \prod_{i=0}^{n-1} \frac{1}{2} \tag{29}$$

in n iterations and divide p by 2 in each iteration. Line 3 performs multiplication. Line 4 makes p even for the reduction where p_0 is the least significant bit of p. Line 5 shifts p to the right by one bit ($p/2$). Line 6 ensures $p < m$. The reason is shown below. In the loop body, p is ensured to be less than $2m$. Then, $p = p + a < 3m$ (line 3), $p = p + m < 4m$ (line 4), and $p = p/2 < 2m$ (line 5). In the finalization, $p = p - m < m$ if $p >= m$ (line 6).

Algorithm 7 MMM (a, b, m) (Montgomery modular multiplication).

inputs: $a = \sum_{i=0}^{n-1} a_i 2^i$, $b = \sum_{i=0}^{n-1} b_i 2^i$, $R = 2^n$, $a, b < m < R$, m: n-bit odd number
output: $p = abR^{-1} \bmod m$
begin
1 $p \leftarrow 0$ /* product */
2 **for** $i = 0$ **to** $n - 1$
3 $p \leftarrow p + b_i a$ /* add multiplicand a to p if $b_i = 1$ */
4 $p \leftarrow p + p_0 m$ /* make p even */
5 $p \leftarrow p \gg 1$ /* $p = p/2$: reduction */
6 **if** $p \geq m$, $p \leftarrow p - m$ /* subtract m from p in the finalization */
7 **return** p
end

Figure 5 shows a possible MMM hardware implementation of Algorithm 7. Red rectangles are registers, and others are combinational circuits. The critical path is the right part that computes the new p in each iteration. It consists of three CPAs and three multiplexers. The most significant bit of the adder output (sign) or the least significant bit p_0 of p can be used as the select signal of multiplexers. Note that $p \gg 1$ can be realized by wiring.

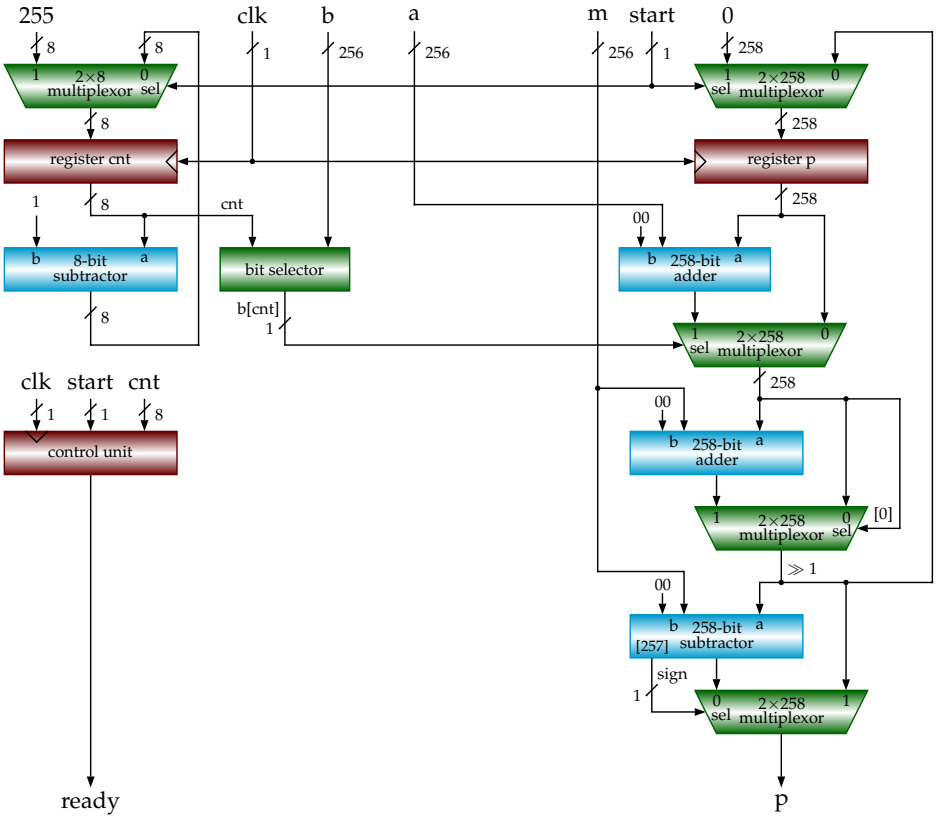

Figure 5. Block diagram of Montgomery modular multiplication (MMM). It implements Algorithm 7.

3.3. Shift-Sub Modular Multiplication Algorithm

The SSMM algorithm [15,16] is formally given in Algorithm 8. It calculates $p = ab \bmod m$, where $a, b < m < 2^n$ and m is an n-bit odd number. It begins with checking the 0th bit of multiplier b. Therefore, in each iteration, the multiplicand u (=a) is shifted to the left by one bit (line 5). Because $a, b < m$, $p + u < 2m$ (line 3), it is enough to subtract m from $p + u$ (line 4), ensuring $p < m$. Because $a, b < m$, $2u < 2m$ (line 5), it is enough to subtract m from $2u$ (line 6), ensuring $u < m$.

Algorithm 8 SSMM (a, b, m) (shift-sub modular multiplication).

inputs: $a = \sum_{i=0}^{n-1} a_i 2^i$, $b = \sum_{i=0}^{n-1} b_i 2^i$, $a, b < m < R$, m: n-bit odd number
output: $p = ab \bmod m$
begin
1 $u \leftarrow a$; $p \leftarrow 0$ /* multiplicand, product */
2 **for** $i = 0$ **to** $n - 1$
3 $p \leftarrow p + b_i u$ /* add multiplicand u to p if $b_i = 1$ */
4 **if** $p \geq m$, $p \leftarrow p - m$ /* subtract m from p */
5 $u \leftarrow u \ll 1$
6 **if** $u \geq m$, $u \leftarrow u - m$ /* subtract m from u */
7 **return** p
end

Figure 6 shows a possible SSMM hardware implementation of Algorithm 8. Red rectangles are registers, and others are combinational circuits. Compared to Figures 4 and 5, here, we reduce the critical path to two CPAs and two multiplexers. The bit b_i (b[cnt] in the figure) is used as a selection signal for the last (bottom) multiplexer. If it is a 0, the value in register p is selected (unchanged). The register u and its corresponding combinational circuits are added for performing lines 5 and 6 in Algorithm 8. Note that the implementation is slightly different from the algorithm. Regardless of b_i, we calculate $p + u$ and $p + u - m$ first. If $p + u - m$ is non-negative, select it; otherwise, select $p + u$. Finally, the value selected by b_i is written to register p. Note that instead of using a multiplexer in the bottom, bit b_i (`b[cnt]` in the figure) can be used as a write enabler for register p.

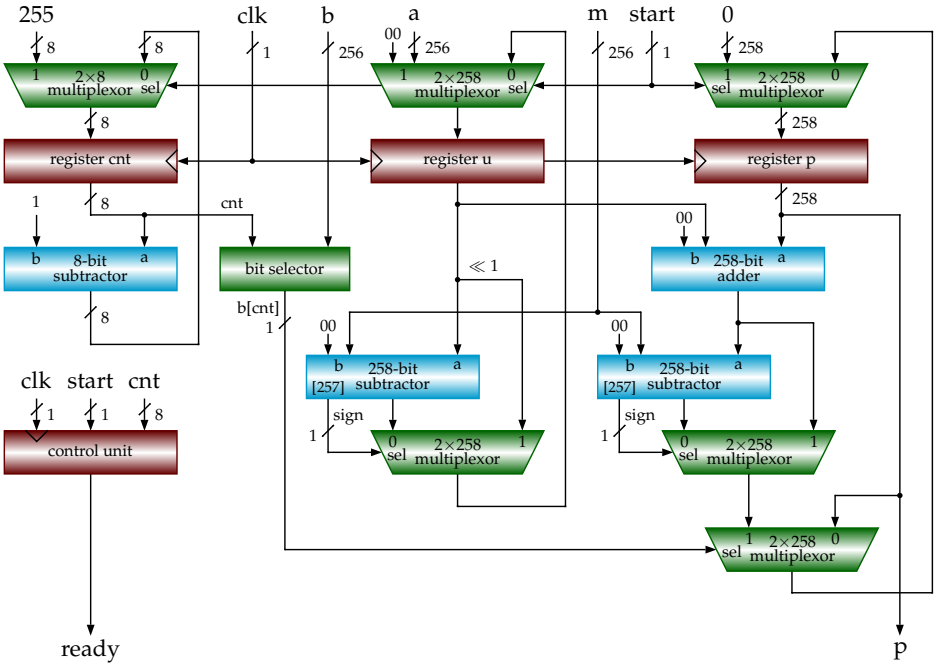

Figure 6. Block diagram of shift-sub modular multiplication (SSMM). It implements Algorithm 8.

3.4. Shift-Sub Modular Multiplication with Advance Preparation Algorithm

From the implementation of the SSMM algorithm, we can see that the critical path is the computation of $p + u - m = p + (u - m)$. m is a modulus and does not change during the multiplication. u is the multiplicand and doubles with each iteration. Then, we can prepare $u - m$ in advance in the previous iteration. u is generated as follows. If $2u' - m$ is non-negative, $2u' - m$ is written to register u; otherwise, $2u'$ is written to register u, where u' is the value of u in the previous iteration. In the current iteration, we have two cases for $u - m$. Case 1: $u - m = 2u' - m$ if $2u' - m$ is negative. Case 2: $u - m = 2u' - m - m = 2u' - 2m$ if $2u' - m$ is non-negative. Then, we just prepare $x = 2u' - m$ and $y = 2u' - 2m$, and store

them in registers. That is, if $x < 0$, $p + (u - m) = p + x$; otherwise, $p + (u - m) = p + y$. The algorithm SSMMPRE is formally given in Algorithm 9.

Algorithm 9 SSMMPRE (a, b, m) (shift-sub modular multiplication with preparation).

inputs: $a = \sum_{i=0}^{n-1} a_i 2^i$, $b = \sum_{i=0}^{n-1} b_i 2^i$, $a, b < m < 2^n$, m: n-bit odd number
output: $p = ab \bmod m$
begin
$u \leftarrow a$; $p \leftarrow 0$; $x \leftarrow 0$; $y \leftarrow a$
for $i = 0$ **to** $n - 1$
  $v \leftarrow p + u$
  if $x < 0$, $w \leftarrow p + x$ /* x: prepared in previous clock cycle */
  else $w \leftarrow p + y$ /* y: prepared in previous clock cycle */
  if $b_i = 1$
    if $w < 0$, $p \leftarrow v$
    else $p \leftarrow w$
  $x \leftarrow 2u - m$; $y \leftarrow 2u - 2m$ /* prepare for use in next clock cycle */
  if $2u < m$, $u \leftarrow 2u$
  else $u \leftarrow 2u - m$
return p
end

Figure 7 shows a possible SSMMPRE hardware implementation of Algorithm 9. Registers x and y hold $2u' - m$ and $2u' - 2m$, respectively, where u' is the value of u in the previous iteration.

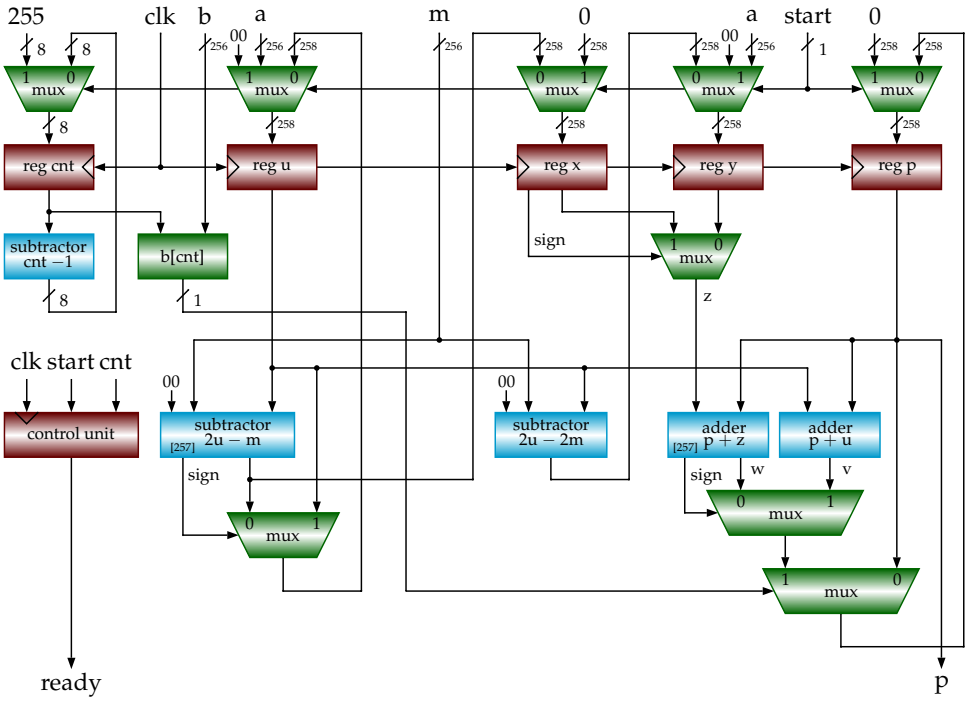

Figure 7. Block diagram of shift-sub modular multiplication with advance preparation (SSMMPRE). It implements Algorithm 9.

3.5. Shift-Sub Modular Multiplication with CSAs and Sign Detection Algorithm

The algorithm SSMMCSA (SSMM with CSAs and sign detection) is formally given in Algorithm 10. Figure 8 shows a possible implementation of SSMMCSA. The two CPAs in Figure 6 are replaced with CSAs. Through our hardware implementation, we find that the part of CSAs that generates c and s is no longer the critical path. We use another register q to store $c + s$ calculated with a CPA, and generate p from q. The critical path of this circuit

contains a CPA and a multiplexer. The output of CSAs consists of carry c and sum s. The result value p will be $(c + s) \bmod m$.

Algorithm 10 SSMMCSA (a, b, m) (shift-sub modular multiplication with CSAs).

inputs: $a = \sum_{i=0}^{n-1} a_i 2^i$, $b = \sum_{i=0}^{n-1} b_i 2^i$, $a, b < m < 2^n$, m: n-bit odd number
output: $p = ab \bmod m$
begin
1 $(c, s) \leftarrow 0$; $u \leftarrow a$
2 **for** $i = 0$ **to** $n - 1$
3 $q \leftarrow c + s$
4 **if** $b_i = 1$
5 $(g, h) \leftarrow \mathrm{CSA}(c, s, u)$ /* add multiplicand u to (c, s) */
6 $(x, y) \leftarrow \mathrm{CSA}(g, h, -m)$
7 **if** sign$(x, y) = 1$ (negative)
8 $(c, s) \leftarrow (g, h)$
9 **else** $(c, s) \leftarrow (x, y)$ /* subtract m from (c, s) */
10 $u \leftarrow u \ll 1$
11 **if** $u \geq m$
12 $u \leftarrow u - m$ /* subtract m from u */
13 **if** $q \geq m$
14 $p \leftarrow q - m$ /* subtract m from q */
15 **return** p
end

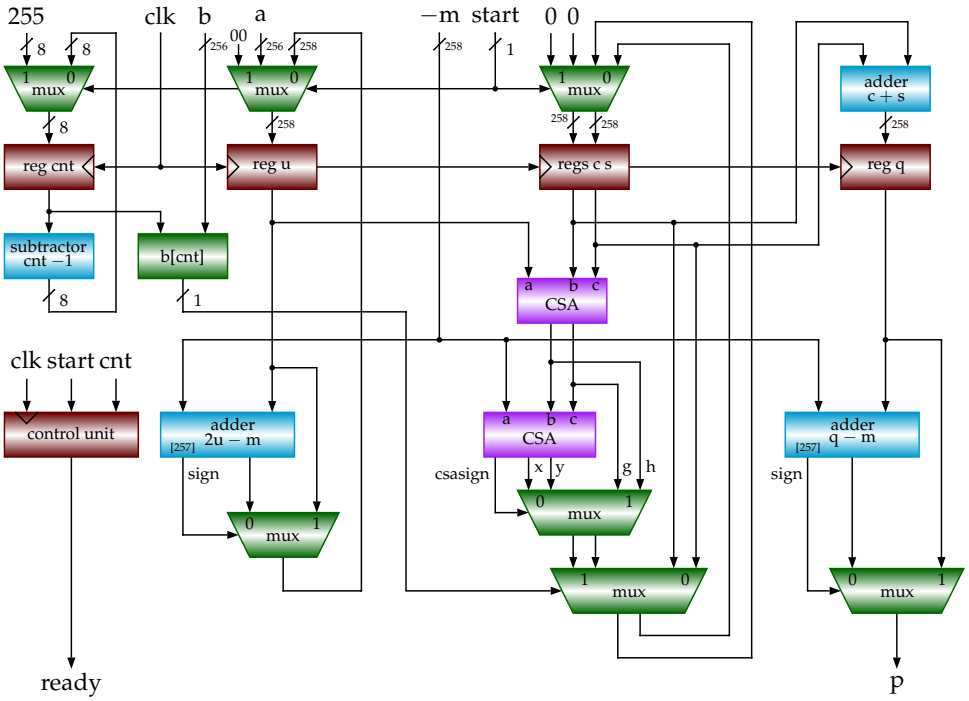

Figure 8. Block diagram of shift-sub modular multiplication with CSAs and sign detection (SSMM-CSA). It implements Algorithm 10.

We will later propose an easy way to determine the sign of $c + s$ from c and s without using CPA to calculate $c + s$. The upper CSA performs $(g, h) \leftarrow \mathrm{CSA}(c, s, u)$ and the other CSA performs $(x, y) \leftarrow \mathrm{CSA}(g, h, -m)$, corresponding to $p + u$ and $p + u - m$ in Figure 6. One CSA's output will be selected with the "csasign" (CSA's sign) signal and stored in the CS register in case $b_i = 1$ (`b[cnt] = 1`).

Note that we can obtain $-m$ from m quickly. Usually $-m = \overline{m} + 1$ where $+1$ needs a CPA. But here, m is an n-bit odd value ($m[0] = 1$), so that we can invert the left $n-1$ bits of m and leave the least significant bit unchanged, that is, $-m = \{\overline{m[n-1:1]}, 1'b1\}$.

Now, we describe how to generate the signal of "csasign". Figure 9 shows an example of determining the sign of $C + S$ from 17-bit C and S for a 16-bit ab mod m when using CSAs. We divide the 17 bits into four windows, namely $W3$, $W2$, $W1$, and $W0$. $W3$ has 5 bits and each of $W2$, $W1$, and $W0$ has 4 bits. We use one 5-bit and three 4-bit CPAs, carry lookahead adders (CLAs), for instance, to perform additions in four windows simultaneously as follows. Note that each of the adders generates a 5-bit result.

$$\begin{aligned} W3[4:0] &= C[16:12] + S[16:12] \quad \text{(Add 5-bit, 5-bit sum)} \\ W2[4:0] &= C[11:8] + S[11:8] \quad \text{(Add 4-bit, 5-bit sum)} \\ W1[4:0] &= C[7:4] + S[7:4] \quad \text{(Add 4-bit, 5-bit sum)} \\ W0[4:0] &= C[3:0] + S[3:0] \quad \text{(Add 4-bit, 5-bit sum)} \end{aligned}$$

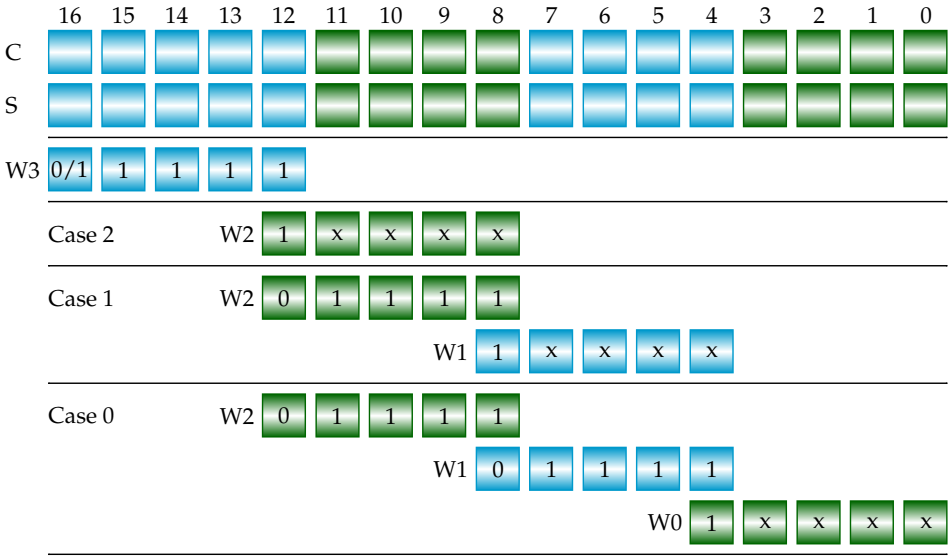

Figure 9. Determining the sign of $P = C + S$ based on the 17-bit outputs C and S of CSAs. We divide 17 bits into four windows. There are five bits in the left-most window. Each of the other three windows has four bits. The C and S are inputs; others are outputs of adders. $W3[4]$ indicates the sign except for Case 0, Case 1, or Case 2, where the sign is the inverse of $W3[4]$ when $W3[3:0] == 4'b1111$.

We summarize the three cases shown in Figure 9 as follows. The signal "sign_inverse" is true, meaning that the original sign ($W3[4]$) is inverted when $W3[3:0] == 4'b1111$.

$$\begin{aligned} \text{sign\_inverse} = {} & (W2[4] == 1) && \text{Case 2} \\ | \ & (W2 == 5'b01111) \ \& \ (W1[4] == 1) && \text{Case 1} \\ | \ & (W2 == 5'b01111) \ \& \ (W1 == 5'b01111) \ \& \ (W0[4] == 1) && \text{Case 0} \end{aligned}$$

The sign bit is a 1 if $P = C + S$ is negative; otherwise, it is 0. When $W3[4]$ is a 0 and "sign_inverse" is true, the sign will be 1 at the condition of $W3[3:0] == 4'b1111$. Similarly, when $W3[4]$ is a 1 and "sign_inverse" is true, the sign will be 0 at the condition of $W3[3:0] == 4'b1111$. Thus, we have the following expression for determining the "csasign".

$$\text{csasign} = W3[4] \oplus (\text{sign\_inverse} \ \& \ (W3[3:0] == 4'b1111)) \tag{30}$$

The symbol $\oplus$ denotes an exclusive OR (XOR) operation. When $W3[3:0] \neq 4'b1111$, the sign equals $W3[4]$, regardless of the case for $W2$, $W1$, or $W0$. Note that $W2$, $W1$, and $W0$ are sums of two 4-bit values c and s, so they do not have the pattern 5'b11111. The maximum sum is 5'b11110 when both c and s have a maximum value of 4'b1111.

In the example described above, the window size is 4. We can let the window size be 2 or 8. Then, we can use eight 2-bit adders or two 8-bit adders to determine the sign of $P = C + S$. The former has a shorter latency introduced by adders, but the logic equation for determining the sign is more complicated. On the other hand, the latter has a simpler logic equation but the adder has a longer latency to determine the sign.

For a 256-bit ECC, there are more options for the number of adder bits and the number of windows. In order to design a low-cost and high-performance circuit, we instigate the cost (the number of ALMs) and performance (the circuit frequency) of the sign detection circuit for using CSAs in 256-bit ECC on FPGA (Field-programmable gate array) chip. There are seven configurations. The experimental results are shown in Table 2 and Figure 10.

Table 2. Cost and performance of sign detection with different configurations. The FPGA chip device is Cyclone V 5CGXFC7D7F31C8.

Windows × Adder Bits	**2 × 128**	**4 × 64**	**8 × 32**	**16 × 16**	**32 × 8**	**64 × 4**	**128 × 2**
ALMs	129	63	33	19	9	7	3
Frequency (MHz)	205.30	248.57	282.09	326.90	342.58	392.46	355.75

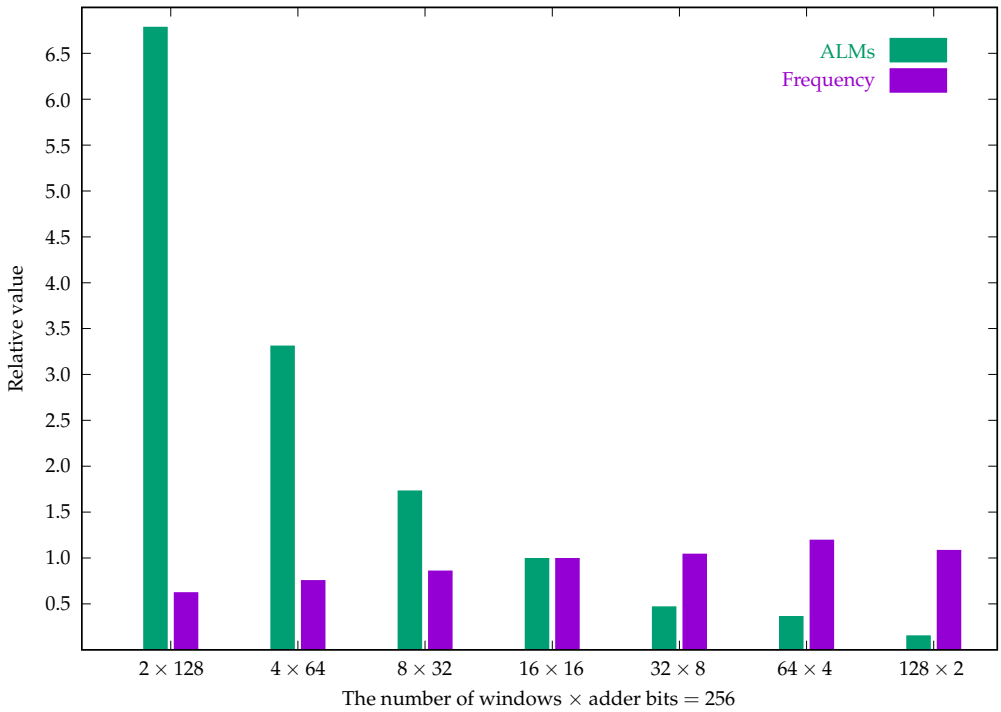

Figure 10. Cost and performance of sign detection with different configurations.

In general, decreasing CPA bits increases the frequency. However, in our simulations, the frequency of the last configuration (128 windows and a 2-bit adder) is lower than the configuration with 64 windows and a 4-bit adder. This is because the logic equation for sign detection becomes complicated. We can also see that larger adders require more ALMs.

Based on the experimental results, our 256-bit ECC implementations use 4-bit adders, so there are 64 windows ($4 \times 64 = 256$). A frequency of 392.46 MHz is measured when the FPGA chip implements only the sign detection circuitry. Implementing both the sign detection circuit and the CSAs results in a frequency of 343.76 MHz, lower than 392.46 MHz. The latency of the CSAs is the same as that of a 1-bit full adder. Clearly, it is less than that of the sign detection circuit. These experimental results imply that the frequency decreases as the circuit becomes larger.

Note that the sign detection circuit itself is a combinational circuit. If we want to test its latency, we can add registers on both the input and output sides. These registers are for

testing purposes only and should be removed for ECC implementations. Otherwise, there will be a delay of two clock cycles, resulting in incorrect timing.

4. Hardware Implementations of Modular Multiplications and ECC

We have implemented modular multiplication algorithms and ECC in Verilog HDL. Unlike sequential program code in software, hardware modules can operate simultaneously. Considering data dependency, we must handle the synchronization between hardware modules using signals such as "start" and "ready". This section describes these implementations and evaluates their cost and performance.

4.1. Hardware Implementations of Modular Multiplications

We have implemented the five modular multiplication algorithms described in the previous section. Since we use the 256-bit key proposed in Secp256k1 [4], the Verilog HDL code for modular multiplication also uses 256-bits. As an example of the Verilog HDL code shown in Appendix A, the circuit calculates $p = ab \bmod m$, where m is a 256-bit odd number and $\{a, b\} < m$, as described in Algorithm 8 (SSMM). The input "start" is a one-clock cycle active signal that tells the module to start modular multiplication. The outputs "ready", "busy", and "ready0" are synchronization signals with other modules. The simulation waveform generated with ModelSim can be found in Figure A1.

When developing Verilog HDL code, it is recommended to use continuous assignment to perform calculations outside of "always" statements that use the clock signal. Verilog HDL code for other modular multiplications can be easily developed by referring to the SSMM example code and the corresponding algorithms and figures.

Table 3 gives the cost performance of the five modular multiplication algorithms. The column of clock cycles shows the required number of clock cycles when executing the modular multiplication algorithm. The column of frequency (MHz) shows the frequency in MHz at which the circuit can work. The column of latency (µs) shows the 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 last column shows the static/dynamic (S/D) power dissipation estimated using the Quartus Prime "Power Analyzer Tool" function when using Intel/Altera Cyclone V 5CGXFC7D7F31C8 FPGA chip.

Table 3. Comparison of modular multiplication algorithms. SSMM is 1.80 times faster than IMM and SSMMCSA is 3.27 times faster than IMM. The FPGA chip device is Cyclone V 5CGXFC7D7F31C8.

Algorithm	Cycles	Freq. (MHz)	Latency (µs)	ALMs	Registers	Power S/D (mw)
IMM	258	35.61	7.25	656	268	358.61/176.66
MMM	258	51.97	4.96	742	277	359.09/162.15
SSMM	258	63.97	4.03	606	527	353.73/117.67
SSMMPRE	258	66.92	3.86	847	1043	354.06/214.35
SSMMCSA	259	116.58	2.22	1117	1048	353.98/161.25

SSMM has a higher frequency than IMM and MMM due to its shorter critical path. IMM only stores the product p, but SSMM needs to store both the product p and the multiplicand u, so SSMM uses about twice as many registers compared to IMM. The number of ALMs used by SSMM is a little bit smaller than that used by IMM. Figure 11 plots the relative cost performance of the modular multiplication algorithms where the cost performance of IMM is set to 1.0, and the data for other implementations are obtained by dividing the corresponding value by the value of IMM. The figure and table show that SSMMCSA provides the highest frequency and lowest latency at the highest hardware cost.

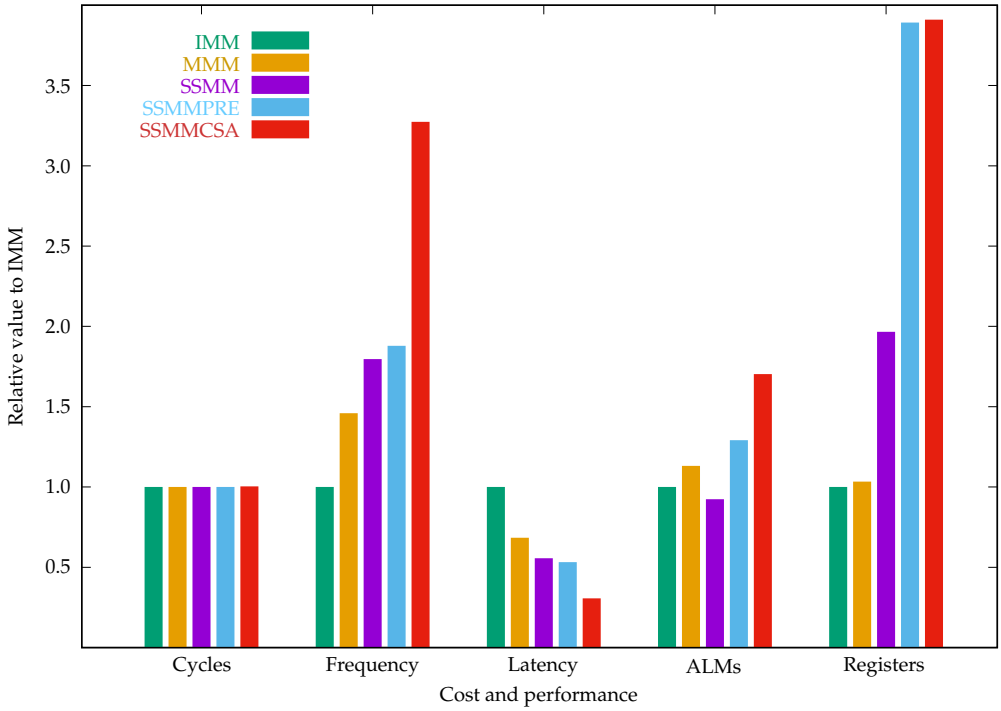

Figure 11. Cost performance comparison of modular multiplication algorithms (set IMM to 1.0).

4.2. Hardware Implementations of ECC

We have implemented ECC in affine, projective, and Jacobian coordinates using the IMM, SSMM, SSMMPRE, and SSMMCSA modular multiplication algorithms. Referring to Algorithm 5, in affine coordinates, the module of scalar point multiplication (`scalarmult`) invokes the module of point addition (`addpoints`) and the module of point doubling (`doublepoint`), as follows (simplified).

```
module scalarmult (clk, rst_n, start, x, y, d, m, a, rx, ry, ready);
    // ...  signal declarations
    addpoints ap (clk, rst_n, start_ap, x1, y1, x2, y2, m, a, apx, apy, ready_ap);
    doublepoint dp (clk, rst_n, start_dp, x1, y1, m, a, dpx, dpy, ready_dp);
    always @(posedge clk or negedge rst_n) begin
        // ...  to generate start_ap and start_dp and register results
        // ...  to check completeness and generate signals for ready
    end
endmodule
```

Based on the `start` signal of module `scalarmult`, we can generate the start signals of `start_ap` for `addpoints` and `start_dp` for `doublepoint`. The result of `scalarmult` is calculated in iterations on the scalar d. Note that in each iteration, `addpoints` and `doublepoint` can be executed in parallel. An important synchronization is to allow `addpoints` to start only after the previous iteration's `doublepoint` has finished.

Referring to Algorithm 5, we use a 256-bit register k to control the iterations. It is initialized with the scalar d at the `start` and shifted one bit to the right at each iteration. After 256 iterations, k becomes 0 and the addition point [`apx`, `apy`] is the result point [`rx`, `ry`]. The modules of scalar point multiplication in projective and Jacobian coordinates invoke the point addition and doubling modules in projective and Jacobian coordinates, respectively. In the `addpoints` and `doublepoint` modules, the result point is calculated based on the algorithms of point addition and point doubling, as described in Algorithm 1 (or 3) and Algorithm 2 (or 4). These modules invoke (1) `modadd` (modular addition), (2) `modsub` (modular subtraction), (3) `modmul` (modular multiplication), and (4) `modinv` (modular inversion). The first two modules are combinational circuits which take one

clock cycle. The last two modules, `modmul` and `modinv`, are sequential circuits for which we must generate the start signals. The Verilog HDL source codes of these two modules are given in the Appendices A and B. Referring to Table 1, in affine coordinates, x of the Q_{ab} and Q_{ba} can be used as the shared secret key for two parties. In projective and Jacobian coordinates, x must be calculated once by X/Z and X/Z^2, respectively. Such calculations can be performed using `modmul` and `modinv`, as shown in the Appendices A and B.

Table 4 gives the cost performance of the ECC in affine, projective, and Jacobian coordinates using the IMM, modular multiplication algorithms proposed in [7–10], SSMM, SSMMPRE, and SSMMCSA. The [7–10] implementations used three-input multiplexers, while the SSMM implementations used only two-input multiplexers. The implementation in [7] used a multiplexer to perform $p + b_i a$. If $b_i = 1$, $p = p + a$. Instead, the implementation in [8] used "AND" gates to perform $p + b_i a$: `p = p + (a & {256{b_i}})`, which is a little slower than the implementation in [7].

As mentioned before, instead of using a multiplexer, bit b_i (`b[cnt]` in Figure 6) can be used as a write enabler for register p. The line labeled "SSMM-WE" shows the cost performance of such an implementation. We can see that it achieves exactly the same cost performance as "SSMM". This is because a DFFE is actually implemented using a DFF and a 2-to-1 multiplexer. The line labeled "SSMM-AND" shows the cost performance of an implementation that uses neither multiplexers nor write-enabled registers. Instead, it uses 256 "AND" gates and a CPA to perform $p + b_i a$. Its performance is lower than using a multiplexer or a write-enabled register. In conclusion, we recommend using a multiplexer for calculating $p + b_i a$, as shown in Figure 6.

Table 4. Comparison of ECC in three coordinates with modular multiplication algorithms.

Algorithm	Cycles	Freq. (MHz)	Latency (ms)	ALMs	Registers	Power S/D (mw)
ECC implementations in affine coordinates						
IMM	402,146	13.50	29.79	14,828	7181	351.63/544.03
[7]	402,146	16.10	24.98	14,790	7046	351.47/511.51
[8]	402,146	15.55	25.86	14,526	6593	351.35/484.90
[9,10]	402,146	16.74	24.02	13,139	7739	351.16/447.03
SSMM	402,146	20.16	19.95	15,096	8354	351.63/543.72
SSMM-WE	402,146	20.16	19.95	15,096	8354	351.63/543.72
SSMM-AND	402,146	16.18	24.85	15,088	8353	351.55/527.63
SSMMPRE	402,146	20.02	20.09	16,782	13,111	352.25/688.02
SSMMCSA	403,166	19.56	20.56	18,372	13,119	352.11/640.69
ECC implementations in projective coordinates						
IMM	396,550	17.17	23.10	30,286	15,333	352.33/694.29
[7]	396,550	21.27	18.64	29,958	14,944	352.30/688.05
[8]	396,550	19.72	20.11	28,467	13,001	352.00/628.22
[9,10]	396,550	21.19	18.71	20,253	13,179	351.05/434.50
SSMM	396,550	29.97	13.23	29,826	23,811	352.24/676.44
SSMM-AND	396,550	20.30	19.53	29,771	23,736	352.18/664.67
SSMMPRE	396,550	29.36	13.51	37,508	42,198	353.50/926.33
SSMMCSA	398,080	21.81	18.25	47,083	48,424	353.46/919.52
ECC implementations in Jacobian coordinates						
IMM	369,079	14.11	26.16	27,861	14,237	351.98/625.50
[7]	369,079	16.63	22.19	27,630	13,561	351.78/583.41
[8]	369,079	16.33	22.60	26,306	11,733	351.71/568.49
[9,10]	369,079	17.63	20.93	19,311	12,124	350.90/403.49
SSMM	369,079	21.23	17.38	28,993	21,862	352.74/777.09
SSMM-AND	369,079	16.44	22.45	28,734	22,296	352.51/730.81
SSMMPRE	369,079	20.75	17.79	35,840	39,724	354.46/1114.60
SSMMCSA	370,502	16.79	22.07	44,624	45,382	354.09/1042.09

The relative cost performance of the ECC implementations in affine, projective, and Jacobian coordinates are plotted in Figures 12, 13 and 14, respectively, where the cost performance of ECC with IMM is set to 1.0, and the data for other implementations are obtained by dividing the corresponding value by the value of the ECC with IMM.

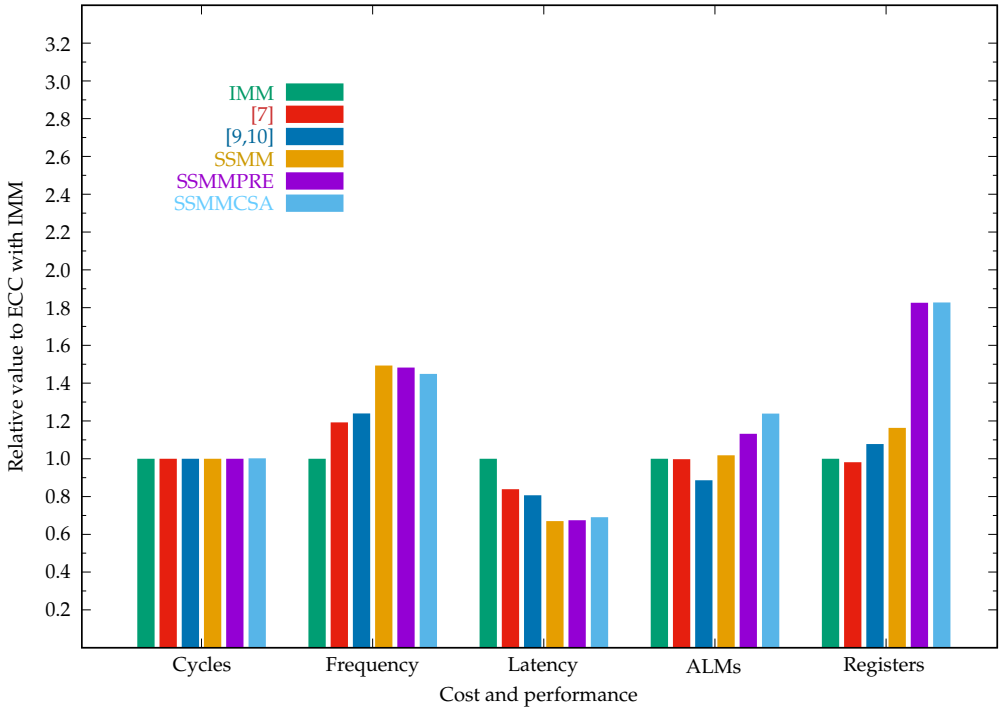

Figure 12. Cost performance comparison of ECC in affine coordinates (set ECC with IMM to 1.0).

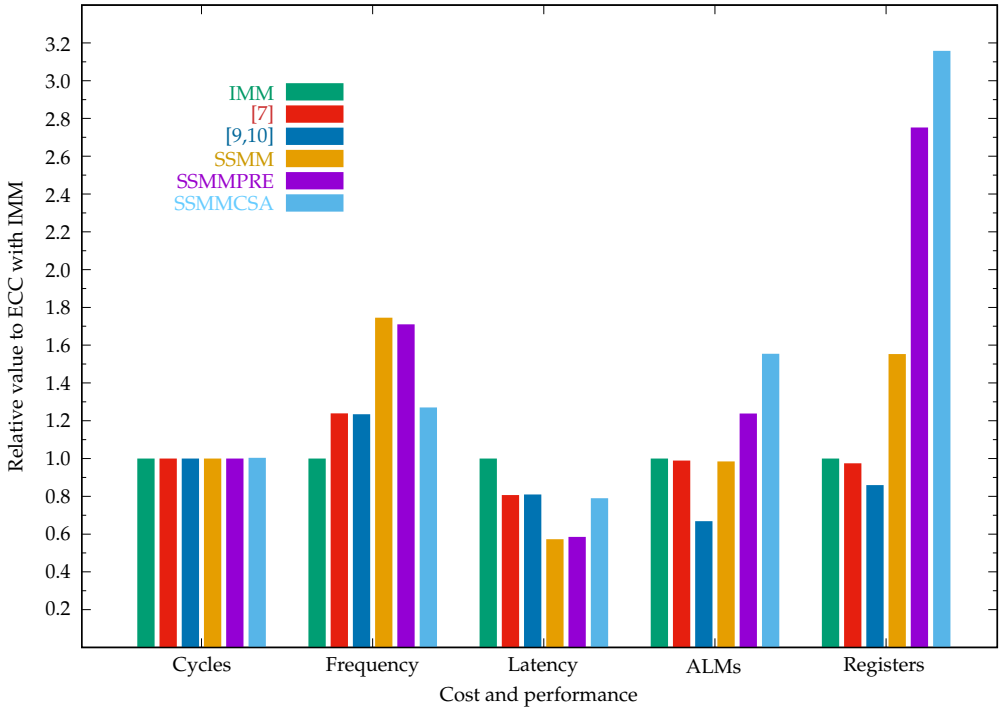

Figure 13. Cost performance comparison of ECC in projective coordinates (set ECC with IMM to 1.0).

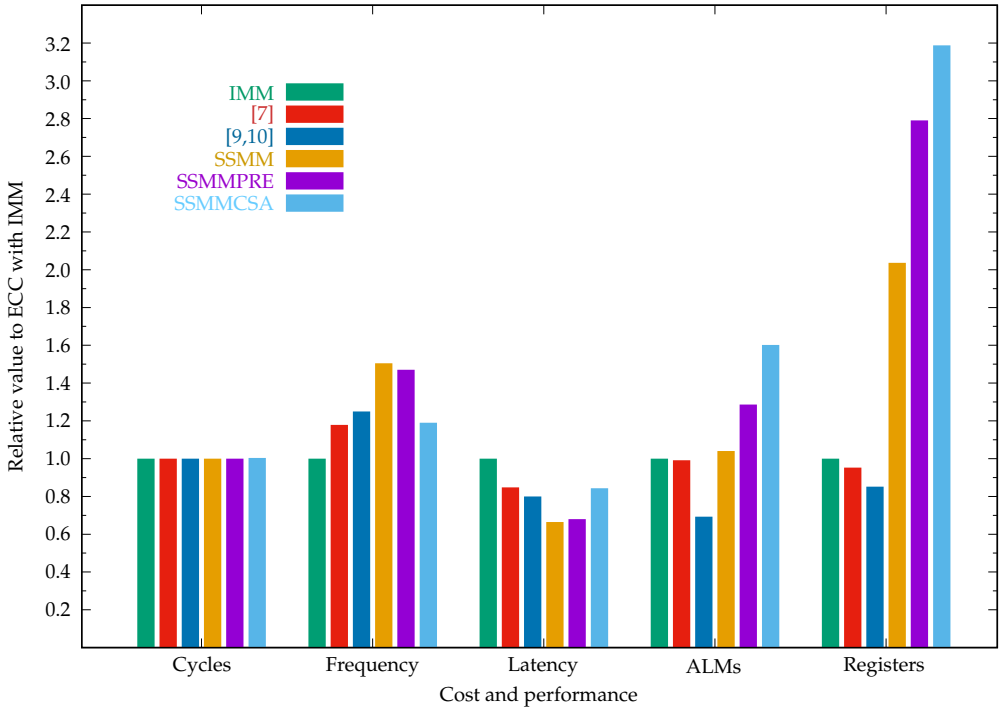

Figure 14. Cost performance comparison of ECC in Jacobian coordinates (set ECC with IMM to 1.0).

The ECC implementations in projective coordinates use more ALMs and registers than those in the other two coordinates. The ECC implementations in affine coordinates have a lower hardware cost than those in the other two coordinates. For all coordinates, the ECC implementations using SSMM, SSMMPRE, and SSMMCSA have higher clock frequencies than those with IMM.

As shown in Table 3, the circuit SSMMCSA has the highest frequency, but applying it to the ECC design does not achieve a higher frequency than the other circuits, such as SSMM. This is because, in ECC design, there are other modules that have longer latency than CSAs and sign detection. For example, `modadd` takes one clock cycle to perform an addition on two operands a and b, a subtraction (subtracting the modulus m from the sum), and a selection using a multiplexer based on the sign of the subtraction result. Another example is `modinv`. Referring to Appendix B, the calculation of $c = ba^{-1} \bmod m$ in modular inversion takes even longer time than `modadd`, as shown as follows, where `r` is the `result` in the source code and `m` is the modulus.

```
if r - 2m >= 0
   c = r - 2m          // c = r - 2m
else if r - m >= 0
        c = r - m      // c = r - 1m
     else if r >= 0
             c = r     // c = r + 0m
          else
             c = r + m // c = r + 1m
          endif
     endif
endif
```

Such calculations take longer time than the CSAs and sign detection operations. Meanwhile, the SSMMCSA circuit is larger than SSMM, this also decreases the frequency.

Figure 15 shows the relative latency of ECC implementations. The value of ECC with IMM is set to 1.0, and the values for other implementations are obtained by dividing the corresponding latency by the latency of the ECC with IMM. We can see that the SSMM in projective coordinates has the lowest latency. This is because, SSMMPRE and SSMMCSA

use a lot of hardware resources, which reduces the frequency. Interestingly, the latency in projective coordinates is lower than that in Jacobian coordinates. The formulas for point addition and point doubling in Jacobian coordinates look simpler than those in projective coordinates, but the calculation of Y_r depends on X_r, resulting in a sequential execution. On the other hand, Y_r and X_r in projective coordinates can be calculated in parallel.

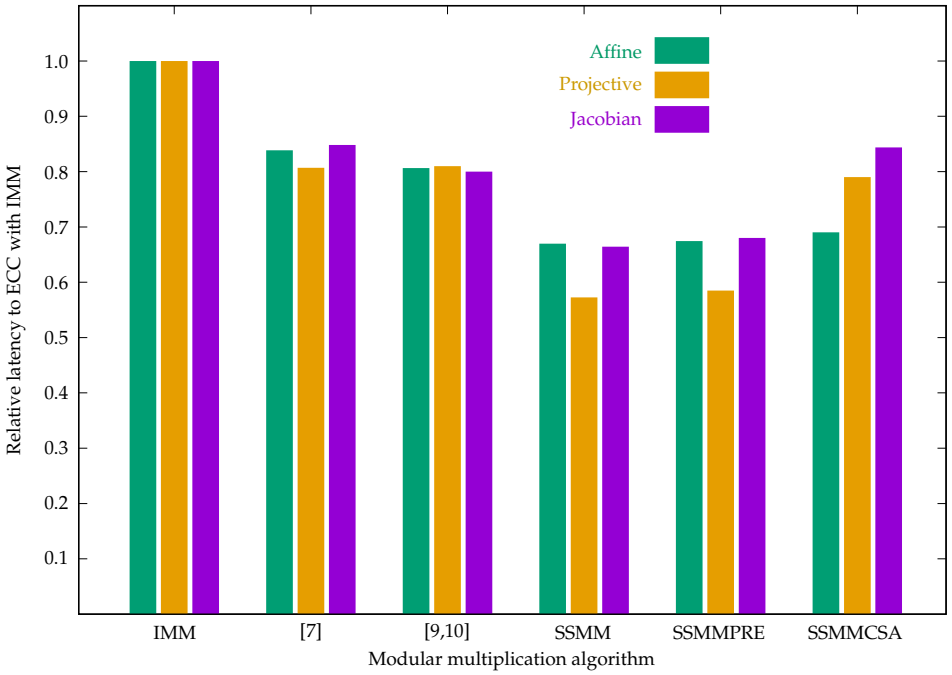

Figure 15. Relative latency of ECC implementations (set the latency of ECC with IMM to 1.0).

The ECC implementations presented in this paper are based on the Secp256k1 curve. It is easy to use the NIST Secp256r1 (P-256) curve [4,19]. We have also implemented the ECC based on the Secp256r1 curve. Our experimental results show that the cost performance of ECC implementations based on both curves is almost the same.

5. Conclusions and Future Work

This paper introduced ECC modular multiplication algorithms and their hardware implementations. Experimental results show that the proposed SSMMPRE (shift-sub modular multiplication with advance preparation) and SSMMCSA (shift-sub modular multiplication with CSAs and sign detection) have lower latencies than SSMM (shift-sub modular multiplication). We proposed a fast and simple method to determine the sign based on the separate output sum and carry of the CSAs (carry save adders). For a 256-bit SSMMCSA, it is recommended to use 64 windows and 4-bit CPAs (carry propagate adders). At the current time, SSMMCSA should be used. We also investigated the ECC implementations in affine, projective, and Jacobian coordinates using the IMM (interleaved modular multiplication), a modified IMM, SSMM, SSMMPRE, and SSMMCSA algorithms. Experimental results show that the ECC implementation in projective coordinates using SSMM has the lowest latency among all the ECC implementations.

The SSMMCSA circuit itself has a higher frequency (116.58 MHz) than SSMM (63.97 MHz), but the ECC circuits using SSMMCSA have the same or even lower frequency than that using SSMM. The next challenge is to find and shorten the critical path to make the SSMMCSA ECC circuit faster than the SSMM ECC circuit. Also, high-radix SSMM algorithms and their use in ECC implementations are worth investigating.

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

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

Appendix A. Verilog HDL Code of Shift-Sub Modular Multiplication (SSMM)

```
‘timescale 1ns/1ns
module modmul (clk, rst_n, start, a, b, m, p, ready, busy, ready0);           // p = a * b mod m
    input          clk, rst_n;
    input          start;
    input  [255:0] a, b, m;
    output [255:0] p;
    output         ready;
    output reg     busy;
    output reg     ready0;
    reg            ready1;
    assign ready = ready0 ^ ready1;
    reg    [257:0] u, s;
    reg      [7:0] cnt;
    wire     [7:0] next_cnt = cnt + 8’d1;
    wire           bi_is_1  = b[cnt];
    wire   [257:0] plus_u   = s + u;                                          // s + u
    wire   [257:0] minus_m  = plus_u - {2’b00,m};                             // s + u - m
    wire   [257:0] new_s    = bi_is_1 ? minus_m[257] ? plus_u : minus_m : s;  // new s
    wire   [257:0] two_u    = {u[256:0],1’b0};                                // 2u
    wire   [257:0] two_u_m  = two_u - {2’b00,m};                              // 2u - m
    wire   [257:0] new_u    = two_u_m[257] ? two_u : two_u_m;                 // new u
    assign p = s[255:0];
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            ready0 <= 0;
            ready1 <= 0;
            busy   <= 0;
        end else begin
            ready1 <= ready0;
            if (start) begin
                u <= {2’b0,a};                                                // u <= a
                s      <= 0;                                                  // s <= 0
                ready0 <= 0;
                ready1 <= 0;
                busy   <= 1;
                cnt    <= 0;
            end else begin
                if (busy) begin
                    s <= new_s;                                               // s <= new_s;
                    if (cnt == 8’d255) begin                                  // finished
                        ready0 <= 1;
                        busy   <= 0;
                    end else begin                                            // not finished
                        u   <= new_u;                                         // u <= new_u;
                        cnt <= next_cnt;                                      // cnt++
                    end
                end
            end
        end
    end
endmodule
```

This code implements Algorithm 8. Its block diagram is shown in Figure 6. The signal “start” is active for one clock cycle to tell the module to start modular multiplication. The signal “ready” remains active for one clock cycle to indicate that the modular multiplication result is available. “ready0” remains active until the multiplier is cleared. This signal is used to confirm the readiness of multiple modules to initiate another module’s operation. Figure A1 shows the simulation waveform generated with ModelSim.

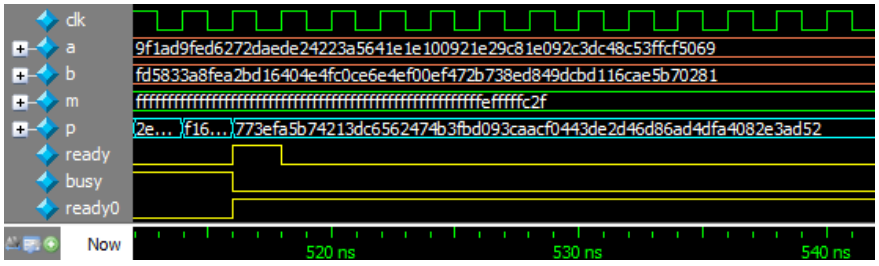

Figure A1. Waveform of SSMM that calculates $p = ab \bmod m$. The Verilog HDL code is given in Appendix A.

Appendix B. Verilog HDL Code of Modular Inversion

```
‘timescale 1ns/1ns
module modinv (clk, rst_n, start, b, a, m, c, ready, busy, ready0);           // c = b * a^{-1} mod m
    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, q, result;
    wire   [259:0] x_plus_m   = x + q;                                        // x + m
    wire   [259:0] y_plus_m   = y + q;                                        // y + m
    wire   [259:0] u_minus_v  = u - v;                                        // u - v
    wire   [259:0] r_plus_m   = result + q;                                   // r + m
    wire   [259:0] r_minus_m  = result - q;                                   // r - m
    wire   [259:0] r_minus_2m = result - {q[258:0],1’b0};                     // r - 2m
    assign c = r_minus_2m[259] ? r_minus_m[259] ? result[259] ? r_plus_m[255:0] :
               result[255:0] : r_minus_m[255:0] : r_minus_2m[255:0];          // c = b * a^{-1} mod m
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            ready0 <= 0;
            ready1 <= 0;
            busy   <= 0;
        end else begin
            ready1 <= ready0;
            if (start) begin
                u <= {4’b0,a};                                                // u <= a
                v <= {4’b0,m};                                                // v <= m
                x <= {4’b0,b};                                                // x <= b
                y <= {260’b0};                                                // y <= 0
                q <= {4’b0,m};                                                // q <= m
                ready0 <= 0;
                ready1 <= 0;
                busy   <= 1;
            end else begin
                if (busy && ((u == 1) || (v == 1))) begin                     // finished
                    ready0 <= 1;
                    busy   <= 0;
                    if (u == 1) begin                                         // if u == 1
                        if (x[259]) begin                                     //     if x < 0
                            result <= x_plus_m;                               //         c = x + m
                        end else begin                                        //     else
                            result <= x;                                      //         c = x
                        end
                    end else begin                                            // else
                        if (y[259]) begin                                     //     if y < 0
                            result <= y_plus_m;                               //         c = y + m
                        end else begin                                        //     else
                            result <= y;                                      //         c = y
                        end
                    end
                end else begin                                                // not finished
                    if (!u[0]) begin                                          // while u & 1 == 0
                        u <= {u[259],u[259:1]};                               //     u = u >> 1
                        if (!x[0]) begin                                      //     if x & 1 == 0
                            x <= {x[259],x[259:1]};                           //         x = x >> 1
                        end else begin                                        //     else
                            x <= {x_plus_m[259],x_plus_m[259:1]};             //         x = (x + m) >> 1
                        end
                    end
                    if (!v[0]) begin                                          // while v & 1 == 0
                        v <= {v[259],v[259:1]};                               //     v = v >> 1
                        if (!y[0]) begin                                      //     if y & 1 == 0
                            y <= {y[259],y[259:1]};                           //         y = y >> 1
                        end else begin                                        //     else
                            y <= {y_plus_m[259],y_plus_m[259:1]};             //         y = (y + m) >> 1
                        end
                    end
                    if ((u[0]) && (v[0])) begin                               // two while loops finished
                        if (u_minus_v[259]) begin                             // if u < v
                            v <= v - u;                                       //     v = v - u
                            y <= y - x;                                       //     y = y - x
                        end else begin                                        // else
                            u <= u - v;                                       //     u = u - v
                            x <= x - y;                                       //     x = x - y
                        end
                    end
                end
            end
        end
    end
endmodule
```

This code is developed based on Algorithm 2.22 in [3]. The signal "start" is active for one clock cycle to tell the module to start modular inversion. The signal "ready" remains active for one clock cycle to indicate that the modular inversion result is available. "ready0" remains active until the multiplier is cleared. This signal is used to confirm the readiness of multiple modules to initiate another module's operation. The testbench is listed below.

```
‘timescale 1ns/1ns
module modinv_tb;
    reg          clk, rst_n, start;
    reg  [255:0] b, a, m;
    wire [255:0] c;
    wire         ready, busy, ready0;
    modinv inst (clk, rst_n, start, b, a, m, c, ready, busy, ready0);
    initial begin
        #0 clk   = 1;
        #0 rst_n = 0;
        #0 start = 0;
        #0 b     = 256’h9cfa1c993911914be0f15bd74a878abe0079c6254b961b82e1abda76387d1d85;
        #0 a     = 256’hd5076ae274e874c2eb0f7778717c39460236549ddd9fc651e68a0c0e787b4ce8;
        #0 m     = 256’hffffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f;
        #1 rst_n = 1;
        #2 start = 1;
        #2 start = 0;
        wait(ready);
        #40 $stop;
    end
    always #1 clk = !clk;
endmodule
```

Figure A2 shows the simulation waveform generated with ModelSim.

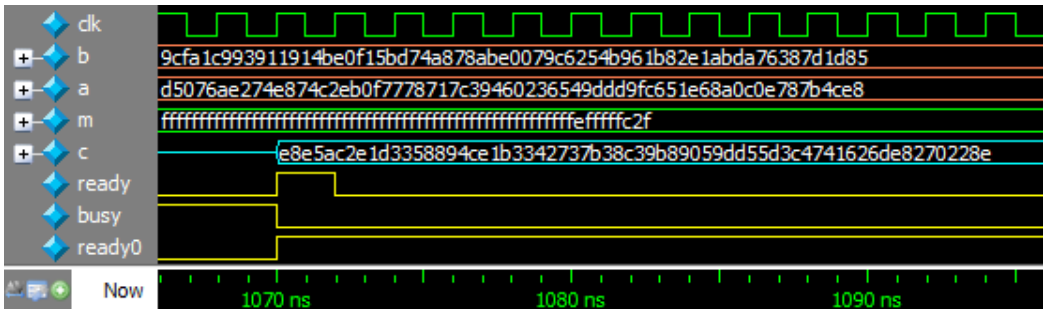

Figure A2. Waveform of modular inversion that calculates $c = ba^{-1} \bmod m$. The Verilog HDL code is given in Appendix B.

References

1. 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 15 October 2023). [CrossRef]
2. 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. 417–426. Available online: https://link.springer.com/content/pdf/10.1007/3-540-39799-X_31.pdf?pdf=inline%20link (accessed on 15 October 2023).
3. Hankerson, D.; Menezes, A.; Vanstone, S. *Guide to Elliptic Curve Cryptography*; Springer: New York, NY, USA, 2004. [CrossRef]
4. Certicom Corp. *Standards for Efficient Cryptography. SEC 2: Recommended Elliptic Curve Domain Parameters*; Certicom Corp: Mississauga, ON, Canada, 2010. Available online: http://www.secg.org/sec2-v2.pdf (accessed on 15 October 2023).
5. 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 15 October 2023).
6. Blakely, G.R. A Computer Algorithm for Calculating the Product AB Modulo M. *IEEE Trans. Comput.* **1983**, *C-32*, 497–500. [CrossRef]
7. Islam, M.M.; Hossain, M.S.; Hasan, M.K.; Shahjalal, M.; Jang, Y.M. FPGA Implementation of High-Speed Area-Efficient Processor for Elliptic Curve Point Multiplication over Prime Field. *IEEE Access* **2019**, *7*, 178811–178826. [CrossRef]
8. Islam, M.M.; Hossain, M.S.; Shahjalal, M.; Hasan, M.K.; Jang, Y.M. Area-Time Efficient Hardware Implementation of Modular Multiplication for Elliptic Curve Cryptography. *IEEE Access* **2020**, *8*, 73898–73906. [CrossRef]
9. Hu, X.; Zheng, X.; Zhang, S.; Cai, S.; Xiong, X. A Low Hardware Consumption Elliptic Curve Cryptographic Architecture over GF(p) in Embedded Application. *Electronics* **2018**, *7*, 104. [CrossRef]

10. Cui, C.; Zhao, Y.; Xiao, Y.; Lin, W.; Xu, D. A Hardware-Efficient Elliptic Curve Cryptographic Architecture over GF(p). *Math. Probl. Eng.* **2021**, *2021*, 8883614. [CrossRef]
11. Hossain, M.S.; Kong, Y.; Saeedi, E.; Vayalil, N.C. High-performance elliptic curve cryptography processor over NIST prime fields. *IET Comput. Digit. Tech.* **2017**, *11*, 33–42. [CrossRef]
12. Liu, Z.; Liu, D.; Zou, X. An Efficient and Flexible Hardware Implementation of the Dual-Field Elliptic Curve Cryptographic Processor. *IEEE Trans. Ind. Electron.* **2017**, *64*, 2353–2362. [CrossRef]
13. Di Matteo, S.; Baldanzi, L.; Crocetti, L.; Nannipieri, P.; Fanucci, L.; Saponara, S. Secure Elliptic Curve Crypto-Processor for Real-Time IoT Applications. *Energies* **2021**, *14*, 4676. [CrossRef]
14. Montgomery, P.L. Modular Multiplication without Trial Division. *Math. Comput.* **1985**, *44*, 519–521. Available online: https://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf (accessed on 15 October 2023). [CrossRef]
15. Li, Y.; Chu, W. Shift-Sub Modular Multiplication Algorithm and Hardware Implementation for RSA Cryptography. In *Hybrid Intelligent Systems*; Springer: Berlin/Heidelberg, Germany, 2021; pp. 541–552. [CrossRef]
16. Li, Y.; Chu, W. Verilog HDL Implementation for an RSA Cryptography using Shift-Sub Modular Multiplication Algorithm. *J. Inf. Assur. Secur.* **2022**, *17*, 113–121. http://www.mirlabs.org/jias/secured/Volume17-Issue3/Paper11.pdf.
17. Bunimov, V.; Schimmler, M. Area and time efficient modular multiplication of large integers. In Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors, The Hague, The Netherlands, 24–26 June 2003; pp. 400–409. [CrossRef]
18. Gayoso Martínez, V.; Hernández Encinas, L.; Sánchez Ávila, C. A Survey of the Elliptic Curve Integrated Encryption Scheme. *J. Comput. Sci. Eng.* **2010**, *2*, 7–13.
19. Chen, L.; Moody, D.; Regenscheid, A.; Robinson, A.; Randall, K. Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters. NIST Special Publication NIST SP 800-186. 2023. Available online: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-186.pdf (accessed on 15 October 2023).