Computer-Aided VLSI System Design

Final Project:

Elliptic Curve Cryptographic Processor

Team 47

R11943004 黃子青

R13943118 林玠志

Table of Content

Chapter 1 APR Results

1.1 Layout

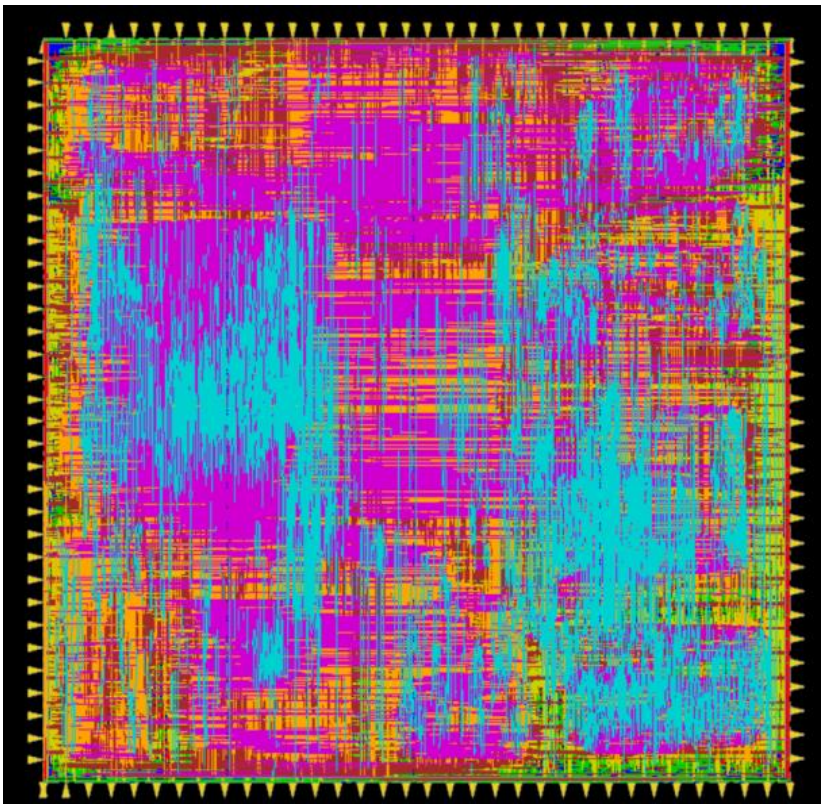

Fig. 1. The snapshot of our final layout

We have organized the APR results into Table 1.

Design Stage	Description	Value
P&R	Number of DRC violations	0
	Number of LVS violations	0
	Die Area (um^2)	2266287.00
	Core Area (um^2)	2205370.71
Post-layout Simulation	Clock Period for Post-layout Simulation	10ns

Table. 1. Our APR result

1.2 DRC/ LVS

```
  VERIFY DRC ..... Sub-Area: {380.800 1332.800 571.200 1503.880} 59 of 64
  VERIFY DRC ..... Sub-Area : 59 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {571.200 1332.800 761.600 1503.880} 60 of 64
  VERIFY DRC ..... Sub-Area : 60 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {761.600 1332.800 952.000 1503.880} 61 of 64
  VERIFY DRC ..... Sub-Area : 61 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {952.000 1332.800 1142.400 1503.880} 62 of 64
  VERIFY DRC ..... Sub-Area : 62 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {1142.400 1332.800 1332.800 1503.880} 63 of 64
  VERIFY DRC ..... Sub-Area : 63 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {1332.800 1332.800 1506.960 1503.880} 64 of 64
  VERIFY DRC ..... Sub-Area : 64 complete 0 Viols.

  Verification Complete : 0 Viols.

*** End Verify DRC (CPU: 0:01:26  ELAPSED TIME: 142.00  MEM: 99.0M) ***
```

Fig. 2. The DRC checking after routing

```
**** 16:52:16 **** Processed 20000 nets.
**** 16:52:17 **** Processed 25000 nets.
**** 16:52:18 **** Processed 30000 nets.
**** 16:52:18 **** Processed 35000 nets.
**** 16:52:19 **** Processed 40000 nets.
**** 16:52:20 **** Processed 45000 nets.
**** 16:52:20 **** Processed 50000 nets.
**** 16:52:21 **** Processed 55000 nets.
**** 16:52:22 **** Processed 60000 nets.
**** 16:52:23 **** Processed 65000 nets.
**** 16:52:23 **** Processed 70000 nets.
**** 16:52:24 **** Processed 75000 nets.
**** 16:52:25 **** Processed 80000 nets.
**** 16:52:26 **** Processed 85000 nets.
**** 16:52:26 **** Processed 90000 nets.
**** 16:52:27 **** Processed 95000 nets.
**** 16:52:28 **** Processed 100000 nets.
**** 16:52:28 **** Processed 105000 nets.
**** 16:52:29 **** Processed 110000 nets.
**** 16:52:30 **** Processed 115000 nets.
**** 16:52:31 **** Processed 120000 nets.
**** 16:52:31 **** Processed 125000 nets.
**** 16:52:32 **** Processed 130000 nets.
**** 16:52:33 **** Processed 135000 nets.
**** 16:52:34 **** Processed 140000 nets.
**** 16:52:34 **** Processed 145000 nets.
**** 16:52:35 **** Processed 150000 nets.
**** 16:52:36 **** Processed 155000 nets.
**** 16:52:37 **** Processed 160000 nets.

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Mon Dec 16 16:52:48 2024
Time Elapsed: 0:00:43.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:24.7  MEM: 264.000M)
```

Fig. 3. The LVS checking after routing

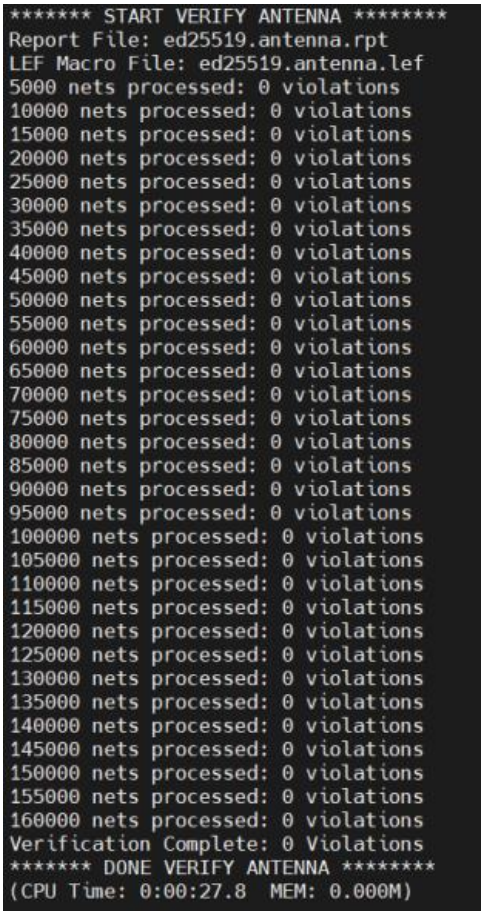

Fig. 4. The antenna checking after routing

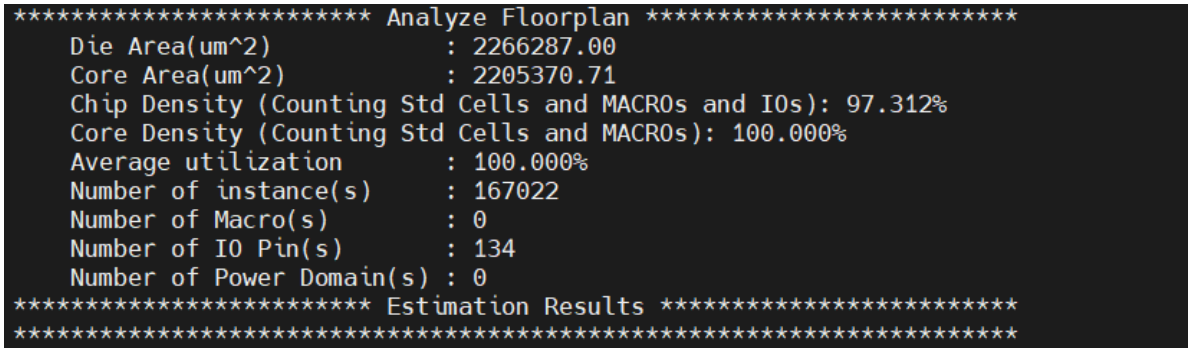

Fig. 5. The snapshot of our final area result

Chapter 2 Algorithm Design

2.1. Scalar Multiplication

Our design for scalar multiplication is based on the algorithm in Fig. 6 described on page 18 of "1131_final_note_v3.pdf".

Algorithm 2: Scalar Multiplication

Parameter: Curve E_{25519}
Input : 255-bit scalar M and point $P = (x_p, y_p) \in E_{25519}$
Output : point $G = (x_G, y_G) = M \times P$
$r = (0, 1, 1)$ // zero point in projective coordinate
$P = (x_P, y_P, 1)$ // point P in projective coordinate
for $i = 255$ to 1 **do**
  $r = r + r$ // point addition
  if ith bit of M is 1 **then**
    $r = r + P$ // point addition
return r

Fig. 6. Algorithm of scalar multiplication

Instead of using the 3 dimensions projective coordinate, we utilize 4 dimensions homogeneous coordinates, extending the representation from (x, y) to (X, Y, Z, T) with X=xZ, Y=yZ, Z=1, T=xyZ. After we finish scalar multiplication, we can reduce X, Y, Z back to x, y by modular division same as using projective coordinate, while coordinate T is no longer needed.

This design choice, inspired by [1], addresses the data dependency issues encountered in the point addition method provided by TA shown in Fig.7. The original approach introduced sequential computation dependencies that hindered the full utilization of the modular multiplication pipeline.

$$(X_1, Y_1, Z_1) + (X_2, Y_2, Z_2) = (X_3, Y_3, Z_3)$$
$$X_3 = Z_1 Z_2 (X_1 Y_2 + X_2 Y_1)(Z_1^2 Z_2^2 - d X_1 X_2 Y_1 Y_2)$$
$$Y_3 = Z_1 Z_2 (Y_1 Y_2 - a X_1 X_2)(Z_1^2 Z_2^2 + d X_1 X_2 Y_1 Y_2)$$
$$Z_3 = (Z_1^2 Z_2^2 - d X_1 X_2 Y_1 Y_2)(Z_1^2 Z_2^2 + d X_1 X_2 Y_1 Y_2)$$

Fig. 7. Point addition under projective coordinate

Using 4 dimensions homogeneous coordinates to calculate point addition can generate lesser intermediate values which requires lesser registers to save them. Besides, it transforms most of the multiplication into add and sub operation which can increase the data parallelism of point addition. On top of that, the computation contains more constants which can be accelerated by pre-computation or using modular adder and subtractor. All of these can help remove the bubbles in two stages pipeline multiplier and effectively reuse our arithmetic logic unit and register file. Later, we would explain how we schedule the computation order to maximize the usage of ALU and minimize the number of registers.

We explicitly explain how to perform point addition and point doubling with 4 dimensions homogeneous coordinate in the below section.

1. Point Addition: For two points P1(X_1, Y_1, Z_1, T_1) and P2(X_2, Y_2, Z_2, T_2), the result of point addition is P3(X_3, Y_3, Z_3, T_3). The computation can be conducted as follows.

$A=(Y_1-X_1)*(Y_2-X_2)$	$E = B – A$	$X_3 = E * F$
$B = (Y_1+X_1)*(Y_2+X_2)$	$F = D – C$	$Y_3 = G * H$
$C = 2d * T_1 * T_2$	$G = D + C$	$Z_3 = F * G$
$D = 2 * Z_1 * Z_2$	$H = B + A$	$T_3 = E * H$

2. Point Doubling : For point P(X_1, Y_1, Z_1, T_1), the result of point doubling is Q(X_2, Y_2, Z_2, T_2). The computation can be conducted as follows.

$A= X_1 * X_1$	$H = B + A$	$X_3 = E * F$
$B = Y_1 * Y_1$	$E = H – D$	$Y_3 = G * H$
$C = 2 * Z_1 * Z_1$	$G = A – B$	$Z_3 = F * G$
$D = (X_1+Y_1)*(X_1+Y_1)$	$F = C + G$	$T_3 = E * H$

To better articulate the benefits brought by the 4D coordinate algorithm, we draw three tables which clearly describe how we schedule the computation order when using 3D coordinate and 4D coordinate. Note that we consider the scenario with only one modular multiplier, one modular adder and one modular subtractor with 8 registers of 256 bits.

- With 3 dimension coordinate point addition :

1. The computation of point addition can be conducted as follows.

$A = X_1 * Y_1$	$E = 2 * A$	$I = d * H$	$M = G + I$
$B = Z_1 * Z_1$	$F = D - E$	$J = B * E$	$X_3 = J * L$
$C = X_1 + Y_1$	$G = B * B$	$K = B * F$	$Y_3 = K * M$
$D = C * C$	$H = A * A$	$L = G - I$	$Z_3 = L * M$

2. The computation order is shown as follows.

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13
mul	A	B	C	H	D	G	F	I	J	K	X3	Y3	Z3	
mod	X	A	B	C	H	D	G	F	I	J	K	X3	Y3	Z3
add			E							M				
sub										L				

Table. 2. The computation order of point addition with 3-dimensional coordinate.

- With 4-dimension coordinate point addition and point doubling:

1. The computation of point addition can be conducted as follows.

$A=(Y_1-X_1)*(Y_2-X_2)$	$E = B - A$	$X_3 = E * F$
$B = (Y_1+X_1)*(Y_2+X_2)$	$F = D - C$	$Y_3 = G * H$
$C = 2d * T_1 * T_2$	$G = D + C$	$Z_3 = F * G$
$D = 2 * Z_1 * Z_2$	$H = B + A$	$T_3 = E * H$

2. The computation order is shown as follows.

cycle	0	1	2	3	4	5	6	7	8
mul	dT2*T1	A	B	X	Z3	X3	T3	Y3	
modq	X	dT2*T1	A	B	X	Z3	X3	T3	Y3
add	Y1+X1	D	C	G	H				
sub	Y1-X1			F	E				

Table. 3. The computation order of point addition with 4-dimensional coordinate.

3. The computation of point doubling can be conducted as follows.

$A = X_1 * X_1$	$H = B + A$	$X_3 = E * F$
$B = Y_1 * Y_1$	$E = H - D$	$Y_3 = G * H$
$C = 2 * Z_1 * Z_1$	$G = A - B$	$Z_3 = F * G$
$D = (X_1+Y_1)*(X_1+Y_1)$	$F = C + G$	$T_3 = E * H$

4. The computation order is shown as follows.

cycle	0	1	2	3	4	5	6	7	8	9
mul	A	B	Z1*Z1	D	X	Y3	Z3	X3	T3	
mod	X	A	B	Z1*Z1	D	X	Y3	Z3	X3	T3
add	X1+Y1			H	C	F				
sub				G		E				

Table. 4. The computation order of point doubling with 4-dimensional coordinate.

These three tables clearly show that the point addition with 3-dimensional coordinate requires more computation steps and generate more intermediate values which need more registers to store them. Also, it contains more multiplication operation, such that most of the computation must wait for the modular multiplier, leaving the modular adder and subtractor idle.

However, if we exploit the expended coordinate, we can reduce the computation steps of point addition from 16 down to 12 and decrease the multiplication operation from 13 down to 8.

The inherent characteristics of the algorithm determine the theoretical upper limit of hardware performance we can achieve, while the practical key lies in planning when to compute each step and where to store the results in which registers. Next, we will describe how each computation step is scheduled and optimized when performing point addition operations using a four-dimensional coordinate system to maximize hardware utilization and minimize the required registers.

- Point addition with 4-dimension expanded coordinate:

$A=(Y_1-X_1)*(Y_2-X_2)$	$E = B - A$	$X_3 = E * F$
$B = (Y_1+X_1)*(Y_2+X_2)$	$F = D - C$	$Y_3 = G * H$
$C = 2d * T_1 * T_2$	$G = D + C$	$Z_3 = F * G$
$D = 2 * Z_1 * Z_2$	$H = B + A$	$T_3 = E * H$

cycle	0	1	2	3	4	5	6	7	8
mul	dT2*T1	A	B	X	Z3	X3	T3	Y3	
modq	X	dT2*T1	A	B	X	Z3	X3	T3	Y3
add	Y1+X1	D	C	G	H				
sub	Y1−X1			F	E				

Fig. 8. ALU computation timeline of point addition

Fig. 8. shows that each of the intermediate values are issued to which arithmetic logic unit at which cycle. Note that the modular multiplier is a two stages pipeline multiplier which performs multiplication at first stage and modular q at second stages. The "X" in the figure represents the bubbles in our 2 stages pipeline modular multiplier.

	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
Reg0	X1	sub1	sub1	A	A	E	E	X3	X3	X3
Reg1	Y1	sum1	sum1	sum1	B	H	H	H	H	Y3
Reg2	Z1	Z1	D	D	F	F	Z3	Z3	Z3	Z3
Reg3	T1	T1	dT1T2	C	G	G	G	G	T3	T3
Reg4	M	M	M	M	M	M	M	M	M	M
Reg5	dT2	dT2	dT2	dT2	dT2	dT2	dT2	dT2	dT2	dT2
Reg6	sub2	sub2	sub2	sub2	sub2	sub2	sub2	sub2	sub2	sub2
Reg7	sum2	sum2	sum2	sum2	sum2	sum2	sum2	sum2	sum2	sum2

Fig. 9. Register update timeline of point addition

Fig. 9. shows that each of the registers would be updated with the intermediate values at which cycle. The slots with blue represent that the registers are going to be updated at that cycle.

For point addition r (X1, Y1, Z1, T1) + p (X2, Y2, Z2, T2) , there exists several places that can be optimized. First, X2, Y2, Z2, T2 would not change during the whole ed25519 computation process. Therefore, Y2-X2 , Y2+X2, dT2 can be precomputed before we start scalar multiplication. As shown in Fig. 9, the value of dT2, sub2(Y2-X2), and sum2(Y2+X2) are already in register 5, 6, and 7. Second, since Z2 is constant 1, thus 2Z1Z2 can be reduced into 2Z1 and calculated by using modular adder. Third, 2*dT1T2 can also be calculated by modular adder. Finally,

since register 0, 1, 2, and 3 contain the intermediate values that are required by X3, Y3, Z3, T3, if we want to store the result X3, Y3, Z3, T3 back to register 0, 1, 2, 3, the order of calculating X3, Y3, Z3, T3 must be carefully designed. As shown in Fig.8. Z3 is calculated first and then is going to occupy register 2 at cycle 6. Therefore, the value X3 that require F must be issued to modular multiplier at cycle 5. The other values Y3, T3 follow the same reason.

- Point doubling with 4-dimension expanded coordinate:

$A = X_1 * X_1$	$H = B + A$	$X_3 = E * F$
$B = Y_1 * Y_1$	$E = H - D$	$Y_3 = G * H$
$C = 2 * Z_1 * Z_1$	$G = A - B$	$Z_3 = F * G$
$D = (X_1+Y_1)*(X_1+Y_1)$	$F = C + G$	$T_3 = E * H$

cycle	0	1	2	3	4	5	6	7	8	9
mul	A	B	Z1*Z1	D	X	Y3	Z3	X3	T3	
mod	X	A	B	Z1*Z1	D	X	Y3	Z3	X3	T3
add	X1+Y1			H	C	F				
sub				G		E				

Fig 10. ALU computation timeline of point doubling

Fig.10 shows that each of the intermediate values is issued to which arithmetic logic unit at which cycle.

	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
Reg0	X1	X1	A	A	H	H	H	H	H	X3	X3
Reg1	Y1	Y1	Y1	B	G	G	G	Y3	Y3	Y3	Y3
Reg2	Z1	Z1	Z1	Z1	$Z1^2$	C	F	F	Z3	Z3	Z3
Reg3	T1	X1+Y1	X1+Y1	X1+Y1	X1+Y1	$(X1+Y1)^2$	E	E	E	E	T3
Reg4	M	M	M	M	M	M	M	M	M	M	M
Reg5	dT2	dT2	dT2	dT2	dT2	dT2	dT2	dT2	dT2	dT2	dT2
Reg6	sub2	sub2	sub2	sub2	sub2	sub2	sub2	sub2	sub2	sub2	sub2
Reg7	sum2	sum2	sum2	sum2	sum2	sum2	sum2	sum2	sum2	sum2	sum2

Fig 11. Register update timeline of point doubling

Fig. 11 shows that each of the registers would be updated with the intermediate values at which cycle. The slots with blue represent that the registers are going to be updated at that cycle.

Point doubling r (X1, Y1, Z1, T1) + r (X1, Y1, Z1, T1) is reduced from point

addition by using the assumption two same points. Therefore, it has lesser computation step compared to point addition. However, because it has data dependency, we must spend one more cycle to compute compared to point addition. Besides, 2* Z1*Z1 can be computed by using modular adder. Note that we try to reuse the registers by carefully scheduling the computation order of X3, Y3, Z3, T3.

2.2 Modular Inversion & Division

After scalar multiplication is completed, point G (X_g, Y_g, Z_g, T_g) need to be reduced into (xg, yg) where xg = X_g/Z_g, yg = Y_g/Z_g. To find out modular inversion of Z_g, we decide to exploit a faster algorithm, inspired from [2], instead of using the algorithm provided in 1131_final_v3_note.pdf.

Algorithm 1: Modular Inversion

Parameter: $q = 2^{255} - 19$
Input : $b \in F_q$
Output : $b^{-1} \mod q \in F_q$
$r = 1$
for $i = 255$ to 1 **do**
  $r = r^2 \mod q$
  if ith bit of $q - 2$ is 1 **then**
    $r = rb \mod q$
return r

Fig. 12. Original algorithm of modular inversion

The algorithm of modular inversion provided in 1131_final_v3_note.pdf has no data parallelism. Each step of multiplication must be performed in sequence, not fully utilizing the 2 stages pipeline modular multiplier. A faster modular inversion in [2] can help eliminate the data dependence in algorithm of Fig. 12.

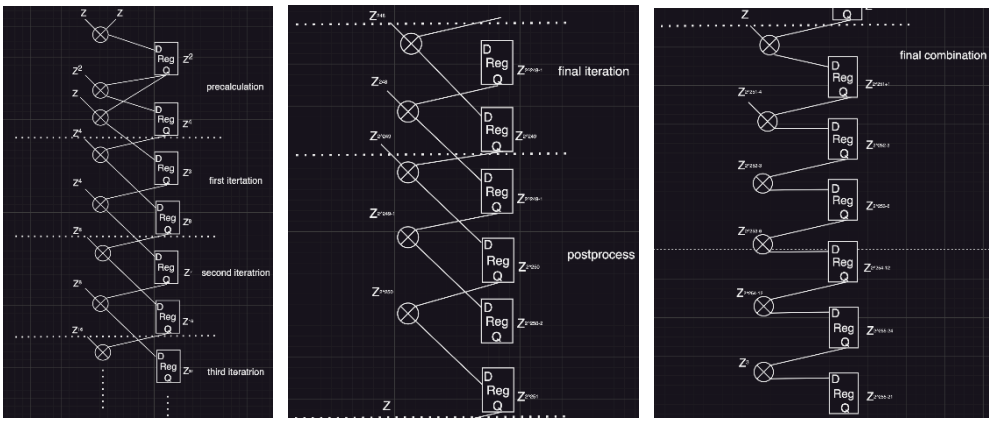

Fig. 13. Fast algorithm of modular inversion.

The flow chart in Fig. 13 shows how to find the modular inversion of a number. Through careful arrangement and design, we can construct the number Z^{-1} (Z^{q-2}). Besides, this method can remove the data dependency of computation and fill up the bubbles in

2 stages pipeline multiplier.

To better illustrate the benefits of this algorithm, we have drawn two tables to compare the fast algorithm with the original algorithm of modular inversion. We can see that the fast modular inversion algorithm eliminates the data dependence in the original algorithm and thus the 2 stages pipeline multiplier can be fully utilized. By deploying this method, we save approximately 500 cycles to calculate the modular inversion of Z.

cycle	0	1	2	3	4	5	6	7	8	9	10	11
mul	r*r	X	r*b	X	r*r	X	r*b	X	r*r	X	r*b	X
modq	X	r*r	X	r*b	X	r*r	X	r*b	X	r*r	X	r*b

Table. 5. The computation order of origin modular inversion algorithm.

cycle	0	1	2	3	4	5	6	7	8	9	10	11
mul	Z^{3}	Z^{8}	Z^{7}	Z^{16}	Z^{15}	Z^{32}	Z^{31}	Z^{64}	Z^{63}	Z^{128}	Z^{127}	Z^{256}
modq	X	Z^{3}	Z^{8}	Z^{7}	Z^{16}	Z^{15}	Z^{32}	Z^{31}	Z^{64}	Z^{63}	Z^{128}	Z^{127}

Table. 6. The computation order of faster modular inversion algorithm.

Chapter 3 Hardware Implementation

Ch3.1 Hardware Architecture

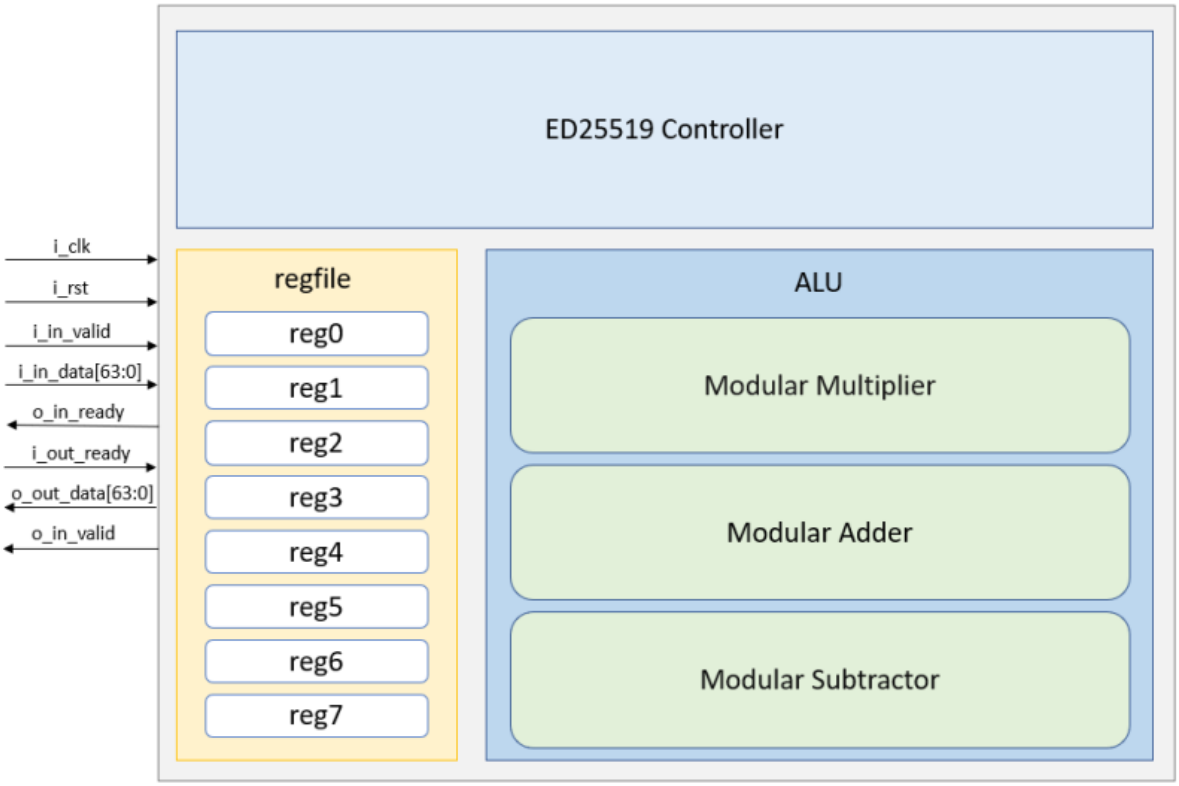

Fig. 14. Hardware architecture of elliptic curve cryptographic processor

Our hardware architecture is shown in Fig.14. It contains a register file with eight 255 bits registers, one modular adder, one modular subtractor, one two-staged pipeline modular multiplier, and a controller.

Ch3.2 Arithmetic Logic Unit

Ch3.2.1 Modular Multiplier

Our modular multiplication algorithm is primarily based on [1]. Below, we have listed the mathematical formulas we organized and derived ourselves:

Step 1: Calculate $x \cdot y$

Let:

$$x = x_1\phi + x_0, \quad y = y_1\phi + y_0$$

Then:

$$x \cdot y = (x_1\phi + x_0)(y_1\phi + y_0) = x_1y_1\phi^2 + (x_1y_0 + x_0y_1)\phi + x_0y_0$$

Simplify:

$$x \cdot y = x_1y_1\phi^2 + \big((x_1 + x_0)(y_1 + y_0) - x_1y_1 - x_0y_0\big)\phi + x_0y_0$$

Notice: We can set $\phi = 2^{129}$ to utilize the property:

$$2^{258} \mod q = 152 \mod q, \quad 2^{255} \mod q = 19 \mod q$$

Step 2: $x \cdot y \mod q$

$$x \cdot y \mod q = \big(x_1y_1\phi^2 + \big((x_1 + x_0)(y_1 + y_0) - x_1y_1 - x_0y_0\big)\phi + x_0y_0\big) \mod q$$

Where:

$$x \cdot y = 2^{258}x_1y_1 + 2^{129}\big((x_1 + x_0)(y_1 + y_0) - x_1y_1 - x_0y_0\big) + x_0y_0$$

Need 3 multiplications!
Considering the worst case is 130 bits (e.g., $(x_1 + x_0)$ or $(y_1 + y_0)$), we use three 130×130 multipliers

Step 3: Modular Reduction

To compute:

$$x \cdot y \mod q = \big(2^{258}x_1y_1 + 2^{129}\big((x_1 + x_0)(y_1 + y_0) - x_1y_1 - x_0y_0\big) + x_0y_0\big) \mod q$$

Using the property:

$$2^{258} \mod q = 152 \mod q$$

We simplify:

$$x \cdot y \mod q = \big(152x_1y_1 + 2^{129}\big((x_1 + x_0)(y_1 + y_0) - x_1y_1 - x_0y_0\big) + x_0y_0\big) \mod q$$

Further using the property:

$$2^{255} \mod q = 19 \mod q$$

Let:

$$152x_1y_1 + 2^{129}\big((x_1 + x_0)(y_1 + y_0) - x_1y_1 - x_0y_0\big) + x_0y_0 = T$$

Define:

$$T_H = T[391:255], \quad T_L = T[254:0]$$

Where:

$$T = T_H \cdot 2^{255} + T_L$$

Then:

$$T \mod q = 19T_H + T_L = t$$

Finally:

$$\text{if } t \geq q, \quad x \cdot y \mod q = t - q$$
$$\text{else } x \cdot y \mod q = t$$

After deriving the mathematical formulas, we found that it is possible to use three 130 x 130 multipliers simultaneously to compute intermediate value x1y1, x0y0, and (x1 + x0)(y1 + y0). The results can then be temporarily stored in a first-stage pipeline register before performing the mod q operation in the next stage. The mod q operation can be efficiently implemented by using shifters to handle the multiplication by 152 and multiplication by 19 , leveraging the modular arithmetic properties of the Ed25519 curve, where $q=2^{255}-19$.

Each 130 x 130 multiplier is implemented using the Karatsuba algorithm [3], as shown in Fig. 15. The Karatsuba-based approach improves efficiency by reducing the number of recursive multiplications, splitting the operands into high and low parts, and combining the partial products through a sequence of additions and subtractions. Specifically, the Karatsuba method reduces the computational complexity from O(n^2) for standard long multiplication to O(n^1.585), which translates to significant savings in both computational time and hardware resource utilization.

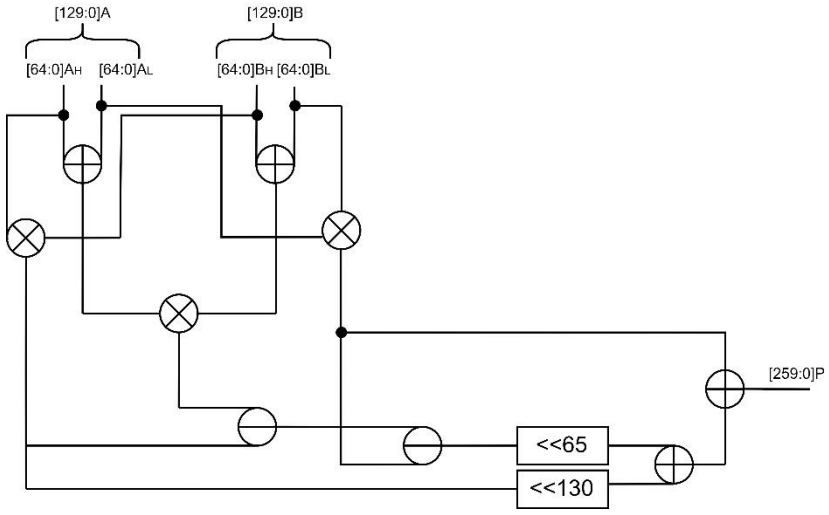

Fig 15. Hardware implementation of 130x130 Karatsuba multiplier.

The hardware architecture illustrated in Fig. 16 integrates these three multipliers into a pipelined modular multiplication unit. At the first stage, the independent multiplications are performed in parallel to compute the three intermediate terms x1y1, x0y0, and (x1+x0)(y1+ y0), ensuring maximum utilization of the hardware resources and minimizing idle time.

The intermediate results are then forwarded to the second stage, where the modular reduction operation is performed. The reduction efficiently incorporates the

property (2^{258} mod q = 152 mod q, 2^{255} mod q = 19 mod q) we mentioned in the step 3 of mathematical formula, applying shifter-based operations to scale the high-order bits and combine them with the low-order terms.

Our approach achieves two major optimizations:

1. Latency Reduction: By parallelizing the multiplications and leveraging the pipelined modular reduction, the overall latency of the operation is minimized, enabling higher throughput for modular multiplication.
2. Resource Efficiency: The Karatsuba-based multipliers, combined with shifter-based reductions, optimize hardware usage, reducing the need for additional multipliers or complex arithmetic units.

Furthermore, the modular reduction step ensures the result remains within the finite field defined by q. After combining the scaled high-order and low-order terms, a conditional subtraction is applied if the result t exceeds q, ensuring the final output satisfies x*y mod q

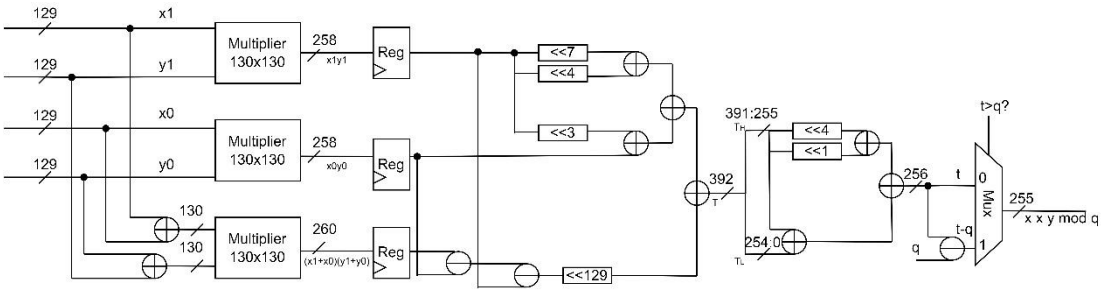

Fig 16. Hardware Implementation of Modular multiplication

Ch3.2.2 Modular adder

Our modular addition and subtraction implementation is based on Fig. 17 described on page 15 of "1131_final_note_v3.pdf". The algorithms leverage conditional operations to ensure the results remain within the finite field Fq.

HW Implementation: Mod. Op. (Add & Sub)

- Let $x, y \in F_q$
- Modular addition:

$$x + y \bmod q = \begin{cases} x + y, if\ x + y < q \\ x + y - q, otherwise \end{cases}$$

- Modular subtraction:

$$x - y \bmod q = \begin{cases} x - y, if\ x \geq y \\ x + (q - y), otherwise \end{cases}$$

Fig. 17. Algorithm of Modular addition

Our modular addition and subtraction implementation is illustrated in Fig.18 and Fig.19 which efficiently handle operations within the finite field Fq. These hardware designs ensure that the results of addition and subtraction remain within the modular range by incorporating comparators**,** adders**,** and multiplexers (MUX)**.**

The modular addition operation computes x + y mod q and ensures the result is bounded by q. Our process can be broken into the following steps:

1. **Addition**:
    - The inputs x and y, each 255 bits, are first added using a standard adder to produce the intermediate result x + y.
2. **Comparison**:
    - A comparator checks whether x + y≥ q. This comparison is crucial to determine if the result exceeds the modular field limit.
3. **Conditional Subtraction**:
    - If x + y≥ q, the value q is subtracted from x + y to bring the result within the modular range.
    - Otherwise, the intermediate result x + y.
4. **Selection (MUX)**:
    - A multiplexer (MUX) selects between x + y and x + y− q based on the comparison result:
        - **0**: If x + y< q, the MUX outputs x + y.
        - **1**: If x + y≥ q, the MUX outputs x + y− q.

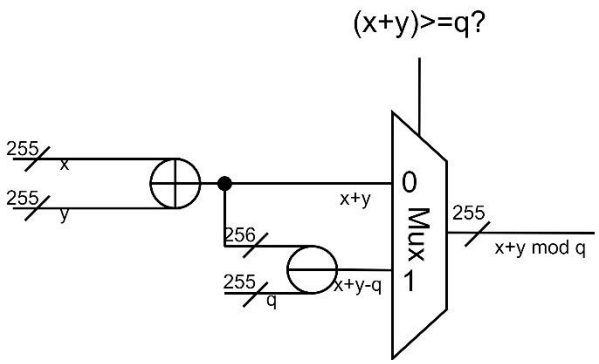

Fig 18. Hardware Implementation of Modular addition

Ch3.2.3 Modular subtractor:

The modular subtraction operation computes $x - y \bmod q$ and handles underflow cases where $x<y$. Our process can be described as follows:

1. **Subtraction**:
    - The inputs x and y are first processed using a standard subtractor to compute $x-y$.
2. **Comparison**:
    - A comparator checks whether $x<y$. This comparison determines if underflow occurs, requiring modular correction.
3. **Conditional Addition**:
    - If $x< y$, the value $q-y$ is added to x to ensure the result remains positive and within the modular range.
    - If $x \geq y$, the result $x-y$ is retained.
4. **Selection (MUX)**:
    - A multiplexer (MUX) selects the correct result based on the comparison result:
        - **0**: If $x \geq y$, the MUX outputs $x-y$.
        - **1**: If $x < y$, the MUX outputs $x+(q-y)$.

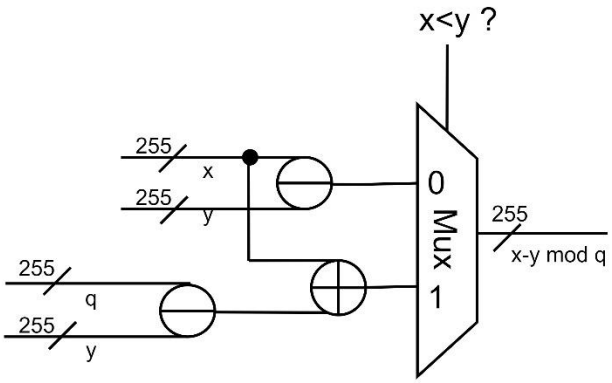

Fig 19. Hardware Implementation of Modular subtraction

-

Ch3.3 ED25519 Controller

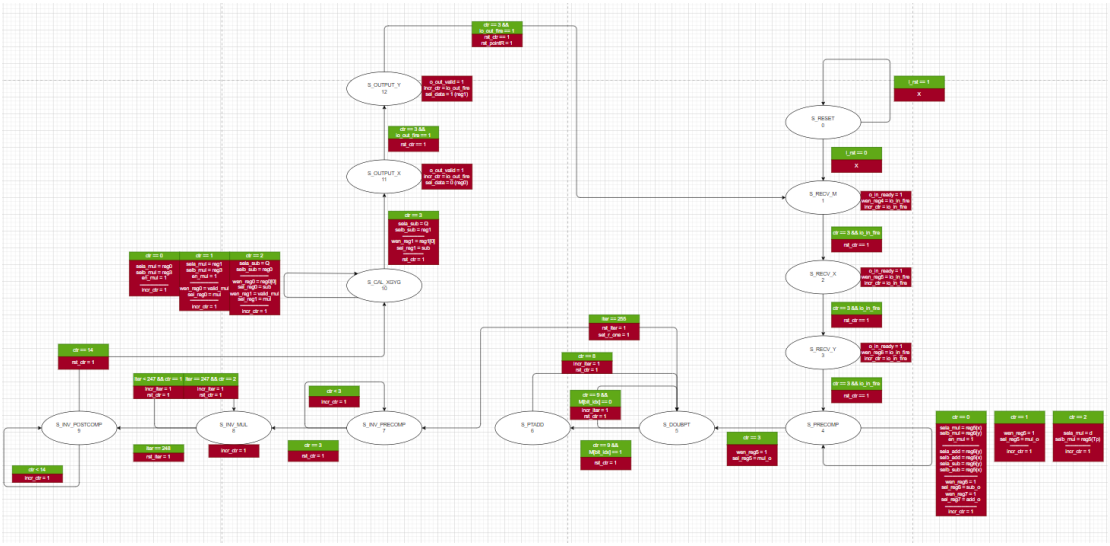

Fig 20. The state transition diagram of ED25519 controller

Our controller is structured into six distinct stages, as illustrated in Fig. 20. The process begins with receiving input data M, Xp, and Yp during the first stage, which includes the statuses S_RECV_M , S_RECV_X, and S_RECV_Y.

In the second stage, precomputations are performed at the S_PRECOMP status. During this stage, we calculate values such as dTp, $Y^2 – X^2$, and $Y^2 + X^2$.

The third stage involves scalar multiplication, executed in the S_DOUBPT and S_PTADD statuses.

Following this, the fourth stage focuses on computing the modular inversion of Z. This is achieved across the statuses S_INV_PRECOMP, S_INV_MUL, and S_INV_POSTCOMP.

In the fifth stage, the X and Y values are reduced back to normal coordinates at the S_CAL_XGYG status.

Finally, in the sixth stage, the results are output during the statuses S_OUTPUT_X and S_OUTPUT_Y.

Chapter 4 Performance Evaluation

V1: Using 4 modular multipliers and 3 adders in Projective Coordinates

Our initial version instantiated four modular multiplier instances and three adders (where we initially combined the adder and subtractor into a single module, using an op signal to determine whether to perform addition or subtraction) to process Scalar Multiplication showed in Fig. 6. This design allowed each cycle to process up to four modular multiplier operations and three modular addition/subtraction operations simultaneously.

The rationale behind this approach was inspired by the use of Projective Coordinates described on page 6 of "1131_final_note_v3.pdf". After careful consideration, we aimed to maximize the operations performed in each cycle to enhance efficiency for point addition, shown in Fig. 21.

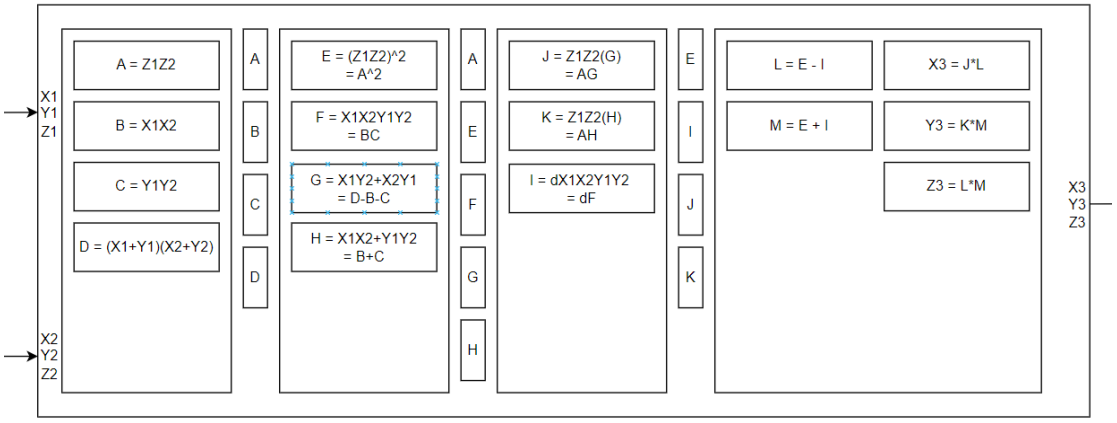

Fig 21. The computation block diagram of origin version

The advantage of this design is that a single point addition can be completed in just 8 cycles (4 stages with each of them 2 cycles). However, the critical drawback is the significantly large area requirement, approximately 7.3 million μm^2. Moreover, due to data dependencies, each modular multiplier must wait one cycle for the previous data to be ready, which prevents the full utilization of pipelining.

V2: Using only 1 modular multiplier and 1 adder and 1 subtractor in extended Projective Coordinates

Referring to [1], which utilizes extended coordinates to effectively leverage pipelining, we separated the modular subtraction from the modular adder into an independent modular subtractor module and rescheduled the computation order as detailed in **Chapter 2.1. Scalar Multiplication**. This optimization significantly reduced the area to 1.725 million μm². The detailed performance metrics are listed in Table 7.

	Area(syn)	Cycle1	Cycle2	Cycle3	Clock	Time1	Time2	Time3	Time	AT
V2(8regs)	1.72564	4695	4740	4785	10	46950	47400	47850	142200	245385.9604

Table 7. Performance of version 2

V3: V2 + fast reduce

Previous versions were based on the modular inversion algorithm provided by the TA, as shown in Fig. 12. Our final version focuses on optimizing the Coordinate Reduction process. The detailed workflow is described in **Chapter 2.2 Modular Inversion & Division**, where we ensured the entire reduction process could fully utilize pipelining, reducing the process by 500 cycles. As a result, the overall workflow also decreased by 500 cycles. The detailed performance metrics are listed in Table 8.

	Area(syn)	Cycle1	Cycle2	Cycle3	Clock	Time1	Time2	Time3	Time	AT
V3(V2+fas	1.808187	4195	4240	4285	10	41950	42400	42850	127200	230001.3864

Table 8. Performance of version 3

We finalized V3 as the final version and performed APR. The resulting APR metrics are shown in Table 9.

	Area(APR)	Cycle1	Cycle2	Cycle3	Clock	Time1	Time2	Time3	Time	AT
V3	2.266287	4195	4240	4285	10	41950	42400	42850	127200	288271.7064

Table 9. Performance of our final version

Reference

[1] Bin Yu, Hai Huang, Zhiwei Liu, Shilei Zhao, Ning Na. "High-performance Hardware Architecture Design and Implementation of Ed25519 Algorithm" [J]. Journal of Electronics & Information Technology, 2021, 43(7): 1821-1827. doi: 10.11999/JEIT200876.

[2] YU B, HUANG H, LIU Z W, et al. High-performance hardware architecture design and implementation of Ed25519 algorithm[J]. Journal of Electronics & Information Technology, 2021, 43(7): 1821-1827.

[3] Yuan, SM., Lee, CY., Fan, CC. (2016). Efficient Digit-Serial Multiplier Employing Karatsuba Algorithm. In: Zin, T., Lin, JW., Pan, JS., Tin, P., Yokota, M. (eds) Genetic and Evolutionary Computing. GEC 2015. Advances in Intelligent Systems and Computing, vol 388. Springer, Cham. https://doi.org/10.1007/978-3-319-23207-2_22