CVSD Final Report
Elliptic Curve Cryptographic Processor

r13k41003 黃逸平

r13943015 張根齊

Team 62

Table of Contents

Introduction

A. Project Description

The Edwards-curve Digital Signature Algorithm (EdDSA) [1] has become a widely adopted standard for data authentication. In this project, we designed an Elliptic Curve Cryptographic Processor to accelerate the core operations of EdDSA. Extended coordinates [2] were utilized for point representation, and a straightforward double-and-add algorithm was implemented for scalar multiplication. As a result, the chip achieves an area of [xxx], an operational frequency of [xxx Hz], and a latency of [xxx cycles].

Overall Design

A. Algorithm Design

1. Algorithm analysis

In the Edwards-curve Digital Signature Algorithm (EdDSA), the overall process is illustrated in **Figure 1**.

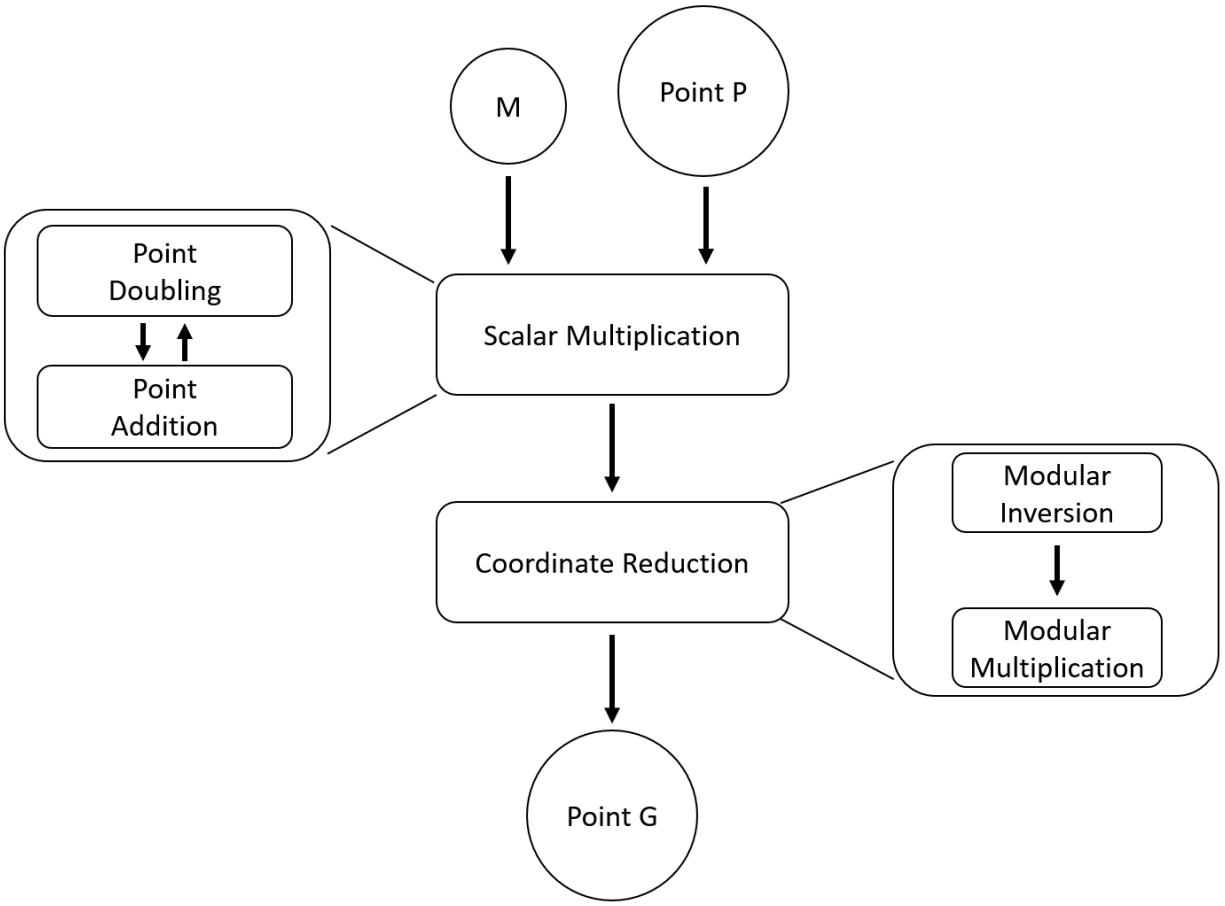

Figure 1. Overview of the Algorithm Flow

- **Scalar multiplication**

Scalar multiplication, a critical operation in the algorithm, can be performed using various methods such as double-and-add, windowed, and sliding-window techniques, among others. For this project, we opted for the simple double-and-add algorithm as it offers the best trade-off between chip area and operational cycles. A detailed analysis of this choice is provided below.

We used Python to estimate the operation cycles of the double-and-add, windowed, and sliding-window techniques, with the detailed comparison presented in **Table 1** (only test PAT0). We estimated the expected operation cycles and number of point doubling and point addition.

To provide a direct comparison, we synthesized a hardware implementation of the windowed algorithm with a window size of 2 alongside the double-and-add algorithm. The results are summarized in **Table 2**. Across the three public test cases, the windowed design reduced the cycle count by an average of 170 cycles per pattern but resulted in an chip area increase of 126705 um^2. This corresponds to a **3.59%** reduction in cycle count but at the cost of a **7.90%** increase in area, highlighting a less favorable trade-off compared to the simpler double-and-add algorithm. Although the windowed techniques achieve fewer operation cycles, this method requires storing additional precomputed lookup table (LUT) values for subsequent point addition and point doubling operations. Furthermore, the increased complexity of ALU input signal selection further contributes to higher hardware costs. As a result, these techniques demand additional sequential cells (LUT registers) and combinational cells (more complex multiplexers), leading to a significant increase in hardware area.

Although the sliding-window algorithm requires fewer LUT entries than the windowed algorithm, we opted not to implement it in hardware due to its significantly increased complexity in input selection logic for the ALU. The highly complex data selection logic would result in a substantial increase in chip area, exceeding even that of the windowed algorithm approach.

Algorithm (size)	Estimated Cycles	Point addition num	Point doubling num
double-and-add	4158	122	255
window (2)	3898	92	256
window (3)	3808	82	256
sliding window (2)	3825	85	255
sliding window (3)	3691	69	256

Table 1. Comparison of Different Scalar Multiplication Algorithms (PAT0)

Algorithm	Total Cycles (PAT 0 ~ 2)	clock period	Area (um^2)	AxT
double-and-add	14235	10ns	1603902	7.61e10
window (size = 2)	13724	10ns	1730607	7.92e10

Table 2. Synthesis Results for Double-and-Add and Windowed Algorithm

- **Point doubling and Point Addition**

In point doubling and point addition, various coordinate representations can be used for the calculations. Among these, projective coordinates and extended coordinates were selected as candidates for evaluation. We scheduled the point doubling and point addition computations for both coordinate representations and simulated the results using Python. Our analysis revealed that extended coordinates significantly outperformed projective coordinates. The detailed comparison is provided in **Table 3**.

When applying the simple double-and-add method for scalar multiplication and basic modular inversion, the total number of addition/subtraction operations across three public test patterns was 8,460 for extended coordinates, compared to 6,888 for projective coordinates. This represents an overall increase of **18.5%** in addition/subtraction operations for extended coordinates.

However, the total number of multiplication operations across the same test patterns was 12,699 for extended coordinates, compared to 25,517 for projective coordinates, demonstrating an overall improvement of **50.2%** in multiplication operations. Given that the complexity of multiplication is significantly higher than that of addition/subtraction, the use of extended coordinates provides greater overall benefits.

As a result, extended coordinates were selected as the coordinate representation for our implementation. The detailed calculation schedules for point doubling and point addition are provided in the **Optimization Techniques** section.

Modular addition, subtraction, and multiplication are utilized in both point doubling and point addition, as well as in the later coordinate reduction process. Modular addition and subtraction are relatively simple and straightforward to implement.

To optimize modular multiplication, we opted not to use Montgomery multiplication. Instead, following the approach in [4], we implemented Karatsuba multiplication combined with mod q module. This approach significantly reduces the cycle count for modular multiplication while also minimizing hardware resource usage. The details of these modules are provided in the **Hardware Implementation - Modules** section.

Coordinate Representation	Estimated add/sub num				Estimated mul num			
	PAT0	PAT1	PAT2	Total	PAT0	PAT1	PAT2	Total
Projective	2266	2296	2326	6888	8947	9052	9157	25517
Extended	2770	2820	2870	8460	4188	4233	4278	12699

Table 3. Comparison of Different Coordinate Representations

- **Modular Operations**
  - **a. Modular Add**

Algorithm: Modular ADD

parameter: $\mathrm{q} = 2^{255} - 19$
input: $X \in F_q, Y \in F_q$
output: $X + Y \bmod \mathrm{q} \in F_q$

if $X + Y < q$
    return $X + Y$
else
    return $X + Y - q$

  - **b. Modular Sub**

Algorithm: Modular SUB

parameter: $\mathrm{q} = 2^{255} - 19$
input: $X \in F_q, Y \in F_q$
output: $X - Y \bmod \mathrm{q} \in F_q$

if $X \geq Y$
    return $X - Y$
else
    return $X + (q - Y)$

c. Modular Mul

When performing Karatsuba multiplication on 255 x 255-bit data, the operation utilizes one 127 x 127 multiplier, one 128 x 128 multiplier, and one 129 x 129 multiplier. These multipliers produce three intermediate results: a 254-bit value H, a 256-bit value L, and a 258-bit value M. The final result of the 255 x 255-bit multiplication is expressed as $2^{256} H + 2^{128}(M-H-L) + L$. This structure can then be leveraged for the subsequent modular reduction (mod q) calculations. The algorithm for modular reduction (mod q) is as follows:

Algorithm: Modular Mul

parameter: $\mathrm{q} = 2^{255} - 19$
input: $X \in F_q$, $Y \in F_q$
output: $X * Y \bmod \mathrm{q} \in F_q$
property: $A * B \bmod \mathrm{q} = (A \bmod \mathrm{q} * B \bmod \mathrm{q}) \bmod \mathrm{q}$
$2^{256} mod\ \mathrm{q} \equiv 38$, $2^{255} mod\ \mathrm{q} \equiv 19$

Karatsuba mul out: H_0, L_0, M_o

(1) $C_h = H_0$, $C_1 = 2^{128}(M_0 - H_0 - L_0) + L_0$
(2) **The first mod q:** $T = 38C_h + C_1$
(3) $T_h = (T_{386}, T_{386}, ..., T_{255})_2$, $T_1 = (T_{254}, T_{254}, ..., T_0)_2$
(4) **The second mod q:** $T' = 19T_h + T1$
(5) **The third mod q:** $T'' = T' - q$
(6) **if** $T'' \leq 0$ return T' **else** return T''

- Coordinate Reduction

In coordinate reduction, the goal is to determine the true coordinates X and Y from their respective coordinate representations. To achieve this, the extended coordinates X and Y must be converted back to their original forms by computing X / Z and Y / Z. In other words, this process involves performing modular inversion and modular multiplication. Specifically, a single modular inversion is required to calculate the inverse of Z (Z^{-1}), followed by modular multiplications with X and Y to derive the final point.

According to Fermat's Little Theorem, $b^{-1} mod\ q = b^{q-2} mod\ q$. Therefore, to compute $Z^{-1} mod\ q$, we need to calculate $Z^{q-2} mod\ q$, where $q = 2^{255-19}$. As a result, the value we want to find is $Z^{2^{255}-21} mod\ q$. Instead of using the algorithm provided by the TA, we were inspired by [3] to develop and schedule our own calculation method for obtaining the final $Z^{2^{255}-21} mod\ q$ value. Our algorithm reduces the total cycle count by 505 compared to the TA-provided algorithm, which has a total cycle count of 4,446. This represents an improvement of 11.4% for PAT0. The detailed schedule is provided in the **Optimization Techniques** section.

2. Optimization techniques

- Point Doubling and Point Addition : 2-stages Modular Mul schedule

The scheduling of operations differs between point doubling and point addition. The regular schedules for point doubling and point addition are presented in **Table 4** and **Table 5**, respectively.

Input: $P(X_1, Y_1, Z_1, T_1), Q(X_2, Y_2, Z_2, T_2)$ **Output:** $Q(X_3, Y_3, Z_3, T_3) = 2P$	
(1)	$A = (X_1)^2$
(2)	$B = (Y_1)^2$
(3)	$t_1 = 2 * (Z_1)^2$
(4)	$t_2 = (X_1 + Y_1)$
(5)	H = A + B
(6)	G = A - B
(7)	F = C + G
(8)	$E = H - t_2^2$
(9)	$X_3 = E * F$
(10)	$Y_3 = G * H$
(11)	$Z_3 = F * G$
(12)	$T_3 = E * H$

Table 4. Schedule of Point Doubling

Input: $P(X_1, Y_1, Z_1, T_1), Q(X_2, Y_2, Z_2, T_2)$ **Output:** $Q(X_3, Y_3, Z_3, T_3) = P + Q$	
(1)	$A = (Y_1 - X_1) * (Y_2 - X_2)$
(2)	$B = (Y_1 + X_1) * (Y_2 + X_2)$
(3)	$C = 2 * d * T_1 * T_2$
(4)	$D = 2 * Z1 * Z_2$
(5)	G = D + C
(6)	F = D - C
(7)	H = B + A
(8)	E = B - A
(9)	$X_3 = E * F$
(10)	$Y_3 = G * H$
(11)	$Z_3 = F * G$
(12)	$T_3 = E * H$

Table 5. Schedule of Point Addition

In our design, we utilized a single modular addition module, a single modular subtraction module, and a single modular multiplication module. The details of the module designs are provided in the Hardware Implementation section. Our modular multiplication module employs a two-stage pipelined design: the first stage performs regular multiplication, and the second stage computes the modular reduction (mod q). To align with this design, we rescheduled the operations for point doubling and point addition. The detailed schedules are presented in **Table 6** and **Table 7**, respectively.

For the point addition operation, since all elements in point Q remain constant throughout the process, certain elements can be precomputed before starting point addition and reused via lookup tables (LUTs) for all subsequent point addition operations. The precomputed LUT details are provided in **Table 8**.

Additionally, some elements can be precomputed at the last cycle of an operation (marked in orange in **Table 6** and **Table 7**) to allow skipping the first cycle calculation in the next point doubling operation.

Cycle	Mul	ModQ	ADD	SUB
(0)	$t_1 = Z_1 * Z_1$			
(1)	$A = X_1 * X_1$	t_1	$t_2 = X_1 + Y_1$	
(2)	$B = Y_1 * Y_1$	A	$C = t_1 + t_1$	
(3)	$t_2 = t_2 * t_2$	B		
(4)		t_2	$H = A + B$	$G = A - B$
(5)	$Y_3 = G * H$		$F = C + G$	$E = H - t_2$
(6)	$Z_3 = F * G$	Y_3		
(7)	$T_3 = F * G$	Z_3		
(8)	$X_3 = F * G$	T_3		
(9)	$t_1 = Z_1 * Z_1$	X_3		

Table 6. Reschedule of Point Doubling (2-stages modular mul)

Cycle	Mul	ModQ	ADD	SUB
(0)	$C = T_1 * T_2$		$B_1 = X_1 + Y_1$	$A_1 = Y_1 - X_1$
(1)	$A = A_1 * A_2$	C	$t_2 = Z_1 + Z_1$	
(2)	$B = B_1 * B_2$	A	$C = C + C$	
(3)		B	$G = t_2 + C$	$F = t_2 - C$
(4)	$Z_3 = F * G$		$H = A + B$	$G = B - A$
(5)	$Y_3 = G * H$	Z_3		
(6)	$X_3 = E * F$	Y_3		
(7)	$T_3 = E * H$	X_3		
(8)	$t_1 = Z_1 * Z_1$	T_3		

Table 7. Reschedule of Point Addition (2-stages modular mul)

Cycle	Mul	ModQ	ADD	SUB
(0)	$T_2 = T_2 * d$			
(1)		T_2	$B_2 = Y_2 + X_2$	$A_2 = Y_2 - X_2$

Table 8. Precalculation of LUT before Point Addition

- **Point Doubling and Point Addition : 3-stages Modular Mul schedule**

To further reduce the cycle time, we designed a three-stage modular multiplication module to minimize the critical path. However, this required rescheduling the operations for point doubling and point addition. The updated schedules for point doubling and point addition using the three-stage modular multiplication module are shown in **Table 9** and **Table 10**, respectively.

We also compared the performance of the synthesis results between the two-stage and three-stage designs. The detailed comparison is presented in **Table 11** (only test PAT0). The three-stage modular multiplication module reduced the clock period to 7ns, but the total execution cycle count increased from 3,941 to 4,444. The synthesis area of the three-stage design is slightly larger than that of the two-stage design, with the three-stage design occupying 1657116 um^2, compared to 1626475 um^2 for the two-stage design.

The area-times-time (AxT) performance of the two-stage design is 5.77e10, while that of the three-stage design is 5.15e10. Overall, the three-stage design demonstrates a 10.74% improvement over the two-stage design.

Cycle	Mul I	Mul II	ModQ	ADD	SUB
(0)	$t_1 = Z_1 * Z_1$				
(1)	$A = X_1 * X_1$	t_1		$t_2 = X_1 + Y_1$	
(2)	$B = Y_1 * Y_1$	A	t_1		
(3)	$t_2 = t_2 * t_2$	B	A	$C = t_1 + t_1$	
(4)		t_2	B		
(5)			t_2	$H = A + B$	$G = A - B$
(6)	$Y_3 = G * H$			$F = C + G$	$E = H - t_2$
(7)	$Z_3 = F * G$	Y_3			
(8)	$T_3 = F * G$	Z_3	Y_3		
(9)	$X_3 = F * G$	T_3	Z_3		
(10)	$t_1 = Z_1 * Z_1$	X_3	T_3		
(11)		t_1	X_3		

Table 9. Reschedule of Point Doubling (3-stages modular mul)

Cycle	Mul I	Mul II	ModQ	ADD	SUB
(0)	$C = T_1 * T_2$			$B_1 = X_1 + Y_1$	$A_1 = Y_1 - X_1$
(1)	$A = A_1 * A_2$	C		$t_2 = Z_1 + Z_1$	
(2)	$B = B_1 * B_2$	A	C		
(3)		B	A	$C = C + C$	
(4)			B	$G = t_2 + C$	$F = t_2 - C$
(5)	$Z_3 = F * G$			$H = A + B$	$G = B - A$
(6)	$Y_3 = G * H$	Z_3			
(7)	$X_3 = E * F$	Y_3	Z_3		
(8)	$T_3 = E * H$	X_3	Y_3		
(9)	$t_1 = Z_1 * Z_1$	T_3	X_3		
(10)		t_1	T_3		

Table 10. Reschedule of Point Addition (3-stages modular mul)

Design	**total cycle**	**clock period**	**area (um^2)**	**AxT**
2-stages MUL	3941	9ns	1626475	5.77e10
3-stages MUL	4444	7ns	1657116	5.15e10

Table 11. Performance comparison of different designs

- **Coordinate Reduction**

Using the TA-provided algorithm, our design requires approximately 1,020 cycles to compute $Z^{2^{255}-21} \, mod \, q$. To optimize this, we adopted the approach described in [3] to reduce the calculation time during the modular inversion stage. However, our overall schedule differs from that of [3], as the final value we aim to compute is not the same. Our calculation schedule is presented in **Figure 2**. With this optimization, our design requires only 3,941 total cycles for the entire process, compared to 4,446 total cycles when using the TA-provided algorithm. (reduce 505 cycles)

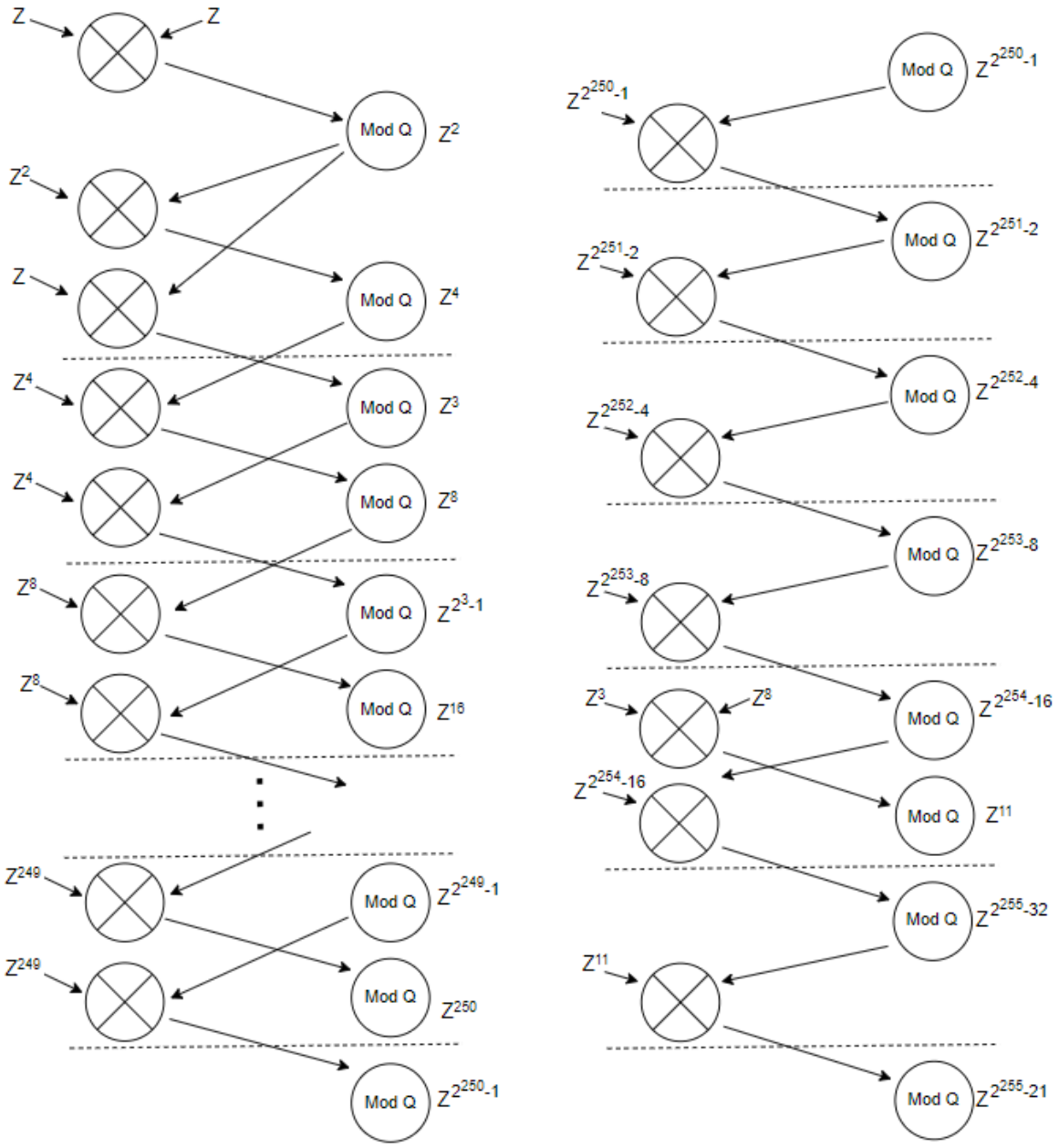

Figure 2. Calculation schedule of modular inversion

B. Hardware Implementation

1. Design Architecture

a. ed25519 top module

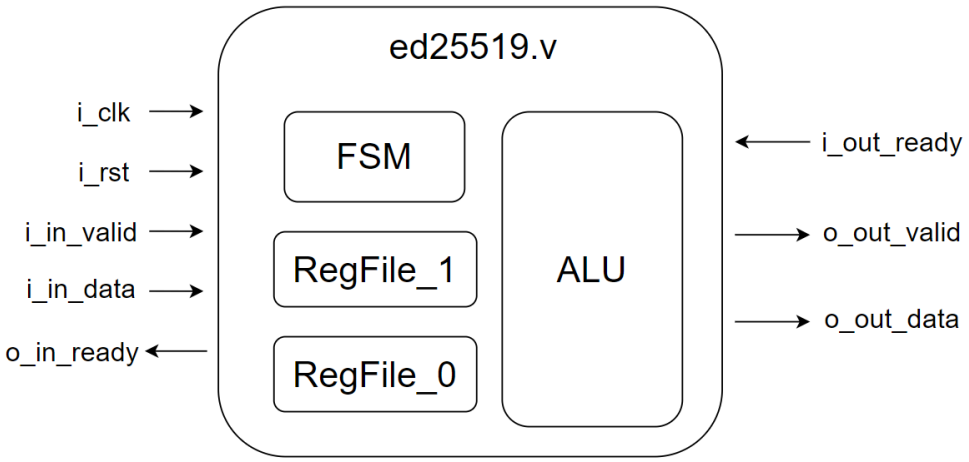

Figure 3. The block diagram of ed25519 top module

b. ALU

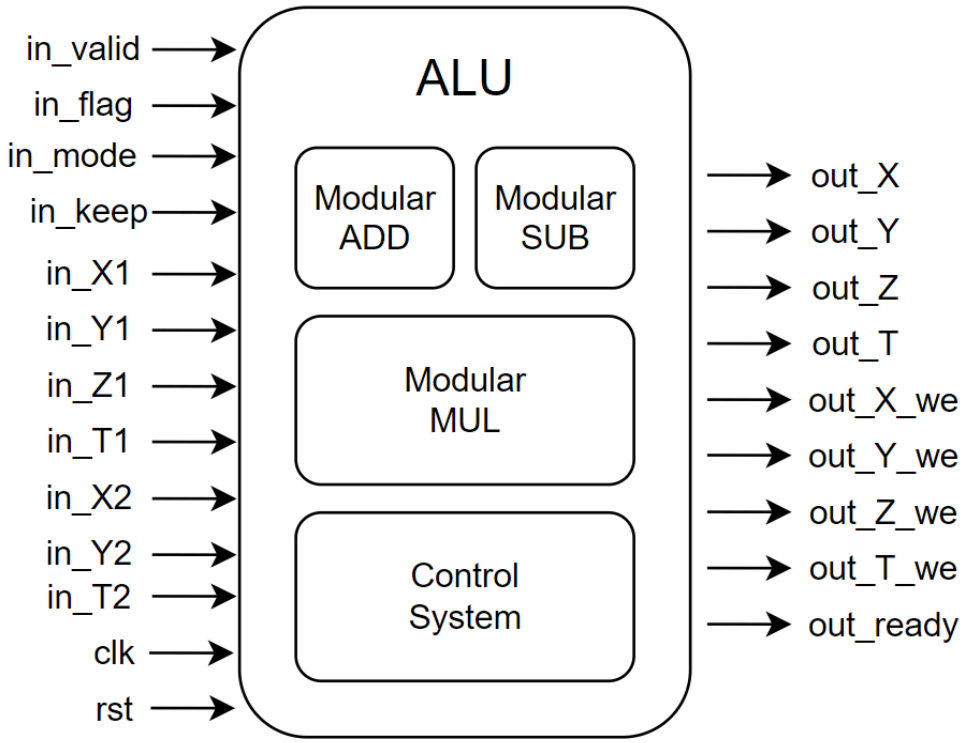

Figure 4. The block diagram of ALU

c. RegFile

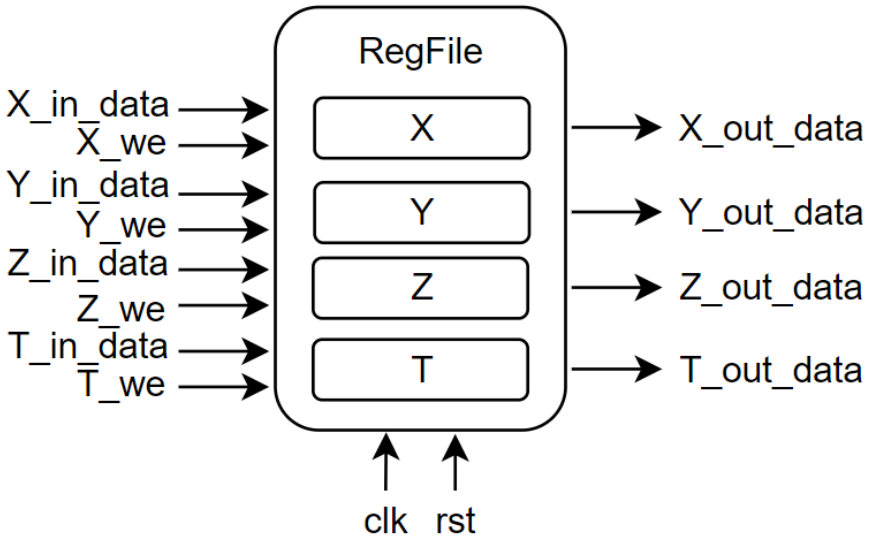

Figure 5. The block diagram of RegFile

d. Modular Mul

i. Mul

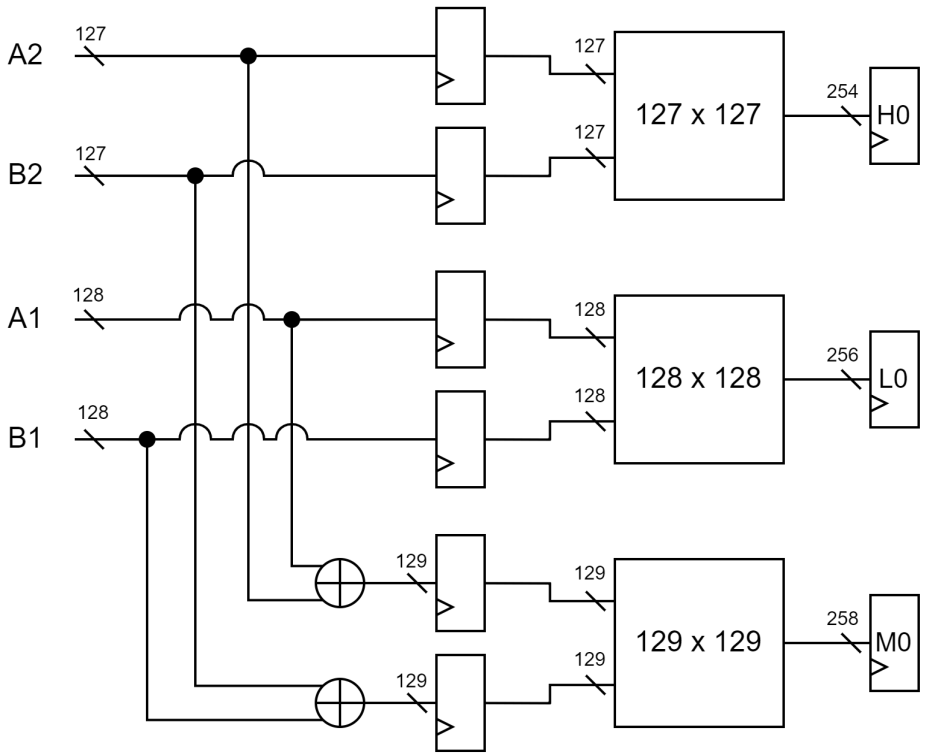

Figure 6. The block diagram of Modular Mul - Mul

ii. ModQ

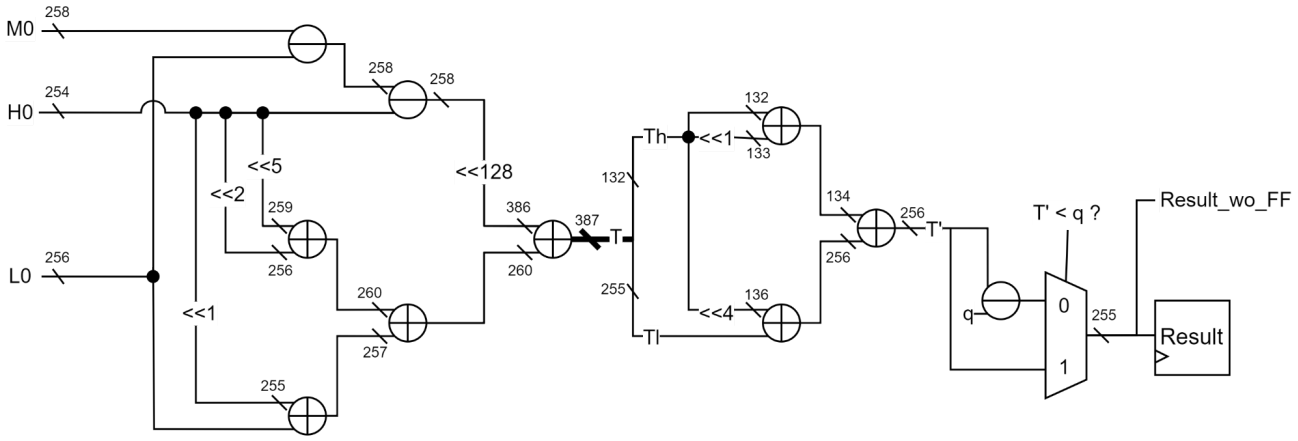

Figure 7. The block diagram of Modular Mul - ModQ

e. Modular Add

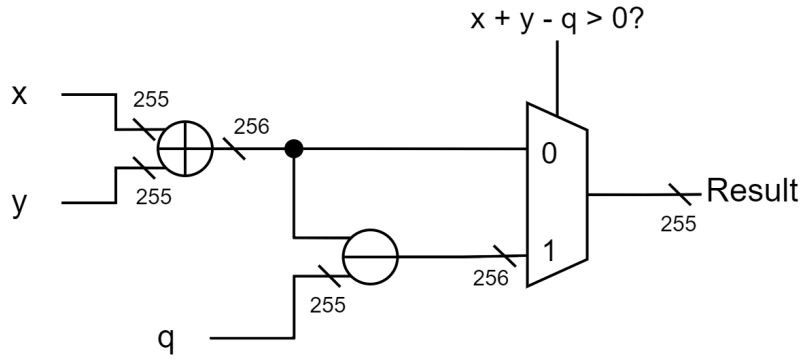

Figure 8. The block diagram of Modular Add

f. Modular Sub

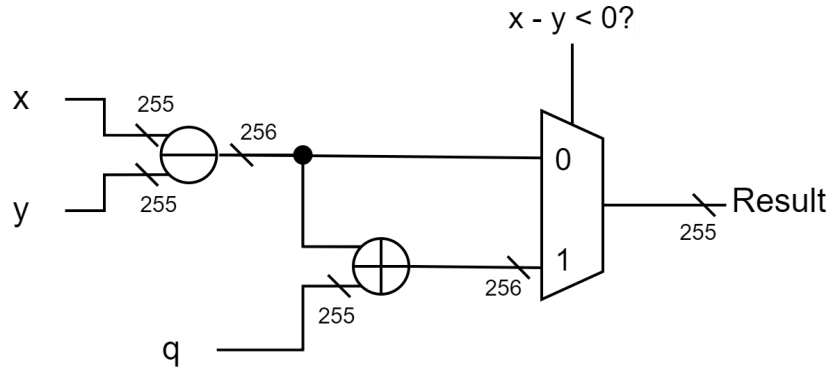

Figure 9. The block diagram of Modular Sub

2. Control system

a. Top module FSM

The top module FSM consists of seven states, with the state flowchart illustrated in **Figure 10**. The details of each state are as follows:

➢ **IDLE**

The top module initializes in the IDLE state upon receiving the reset signal. In the next cycle, it transitions to the LOAD state, where it awaits input data.

➢ **LOAD**

In the LOAD state, when i_in_valid is high, the input data is loaded into the register file. Loading requires 4 cycles for M, 4 cycles for X, and 4 cycles for Y. However, the total loading cycle count may not always equal 12, as the i_in_valid signal is random. Input data is loaded only when i_in_valid is high. Once M, X, and Y are successfully loaded into the register file, the FSM transitions to the PRECAL state.

➢ **PRECAL**

In the PRECAL state, the ALU performs pre-calculations for all elements required in subsequent operations. Since we use extended coordinates to represent points, the Z and T coordinates must be initialized. The Z coordinate is always initialized to 1, so no calculation is necessary for it. However, the T coordinate, which is $X \times Y$, needs to be computed. Additionally, to optimize the point addition operation, some lookup table (LUT) values are precomputed to reduce calculation cycles. This technique is discussed in detail in the **Algorithm Design - Optimization Techniques** section. Once all pre-calculations are complete, the FSM transitions to the MUL state to perform scalar multiplication.

➢ **MUL**

In the MUL state, the ALU performs up to 255 point doubling operations and up to 255 point addition operations, as dictated by the simple double-and-add algorithm used for scalar multiplication. The exact number of point addition operations depends on the value of M. An 8-bit counter increments by 1 after each set of point doubling and point addition is completed. Once the counter reaches 255, the FSM transitions to the DIV state to perform the modular inversion.

➢ **DIV**

In the DIV state, the ALU performs a modular inversion on the Z coordinate obtained from the scalar multiplication to compute the inverse of Z. Once the inversion of Z has been calculated, the FSM transitions to the REDUCE state.

➢ **REDUCE**

In the REDUCE state, the ALU calculates X / Z and Y / Z to convert the extended coordinates back to the standard coordinate representation. Additionally, the design

checks whether the final coordinates X and Y are both even. If not, the corresponding even values for X and Y are computed. Finally, once the final point $G(X, Y)$ is prepared, the FSM transitions to the OUT state for output.

➢ **OUT**

In the OUT state, the design sets o_out_valid to high and waits to output the X and Y coordinate data. Each X and Y is a 255-bit value, padded with zeros at the most significant bits (MSB) to align with the output format. The processor outputs 64 bits of data per cycle when i_out_ready is high, requiring 8 cycles to complete the output process. Once all the data has been transmitted, the FSM transitions back to the IDLE state.

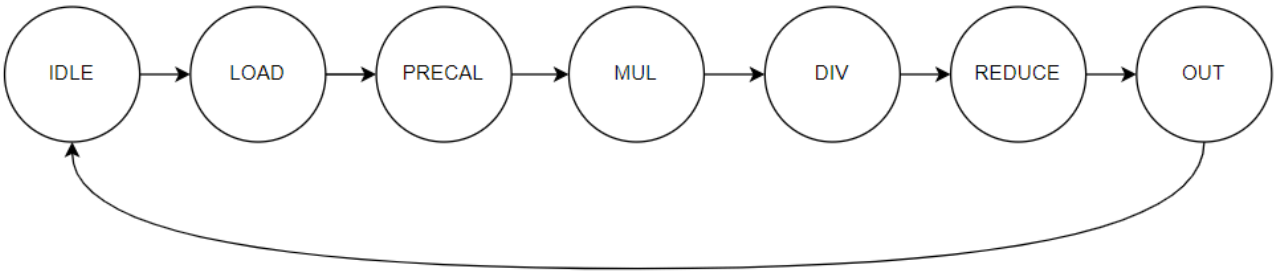

Figure 10. Top module FSM

b. ALU FSM

The ALU FSM consists of six states, with the state flowchart illustrated in **Figure 11**. The details of each state are as follows:

➢ **IDLE**

The ALU is initialized in the IDLE state. Upon receiving in_valid and in_mode signals from the top module, the FSM transitions to the corresponding mode based on the value of in_mode. Each mode is designed for a specific calculation task. The DOUBLE and ADD modes handle scalar multiplication, while the MUL mode is optimized for the pre-calculation of extended coordinates and lookup tables (LUTs), as previously described. The DIV mode is responsible for performing modular inversion, and the REDUCE mode executes the coordinate reduction process. The details of each mode's operation are described below.

➢ **DOUBLE**

In the DOUBLE state, the ALU performs the point doubling operation as part of scalar multiplication. The top module sends an in_flag signal to the ALU, which is determined by the value of M. If in_flag is high, the FSM transitions to the ADD

state; otherwise, it remains in the DOUBLE state for the next cycle. At the end of the scalar multiplication process, the top module sets the in_keep signal to low and sends it to the ALU. This indicates the end of scalar multiplication, prompting the ALU to transition back to the IDLE state.

➢ **ADD**

In the ADD state, the ALU performs point addition operations as part of the scalar multiplication process. Similar to the DOUBLE state, the ALU transitions back to the IDLE state at the end of the scalar multiplication process. Otherwise, it returns to the DOUBLE state to continue the computation.

➢ **MUL**

The MUL state is specifically designed for pre-calculation tasks, as all pre-calculation operations require multiplication. In this state, the ALU performs a single multiplication operation and then transitions back to the IDLE state.

➢ **DIV**

In the DIV state, the ALU performs the modular inversion of the Z data. The details of this modular inversion calculation have been discussed earlier in the **Algorithm Design - Optimization Techniques** section. Once the calculation is done, it returns to the IDLE state.

➢ **REDUCE**

In the REDUCE state, the ALU calculates $X * Z^{-1}$ and $Y * Z^{-1}$, and prepares the even point values for X and Y for later use. Once all calculations are complete, the FSM transitions to the IDLE state.

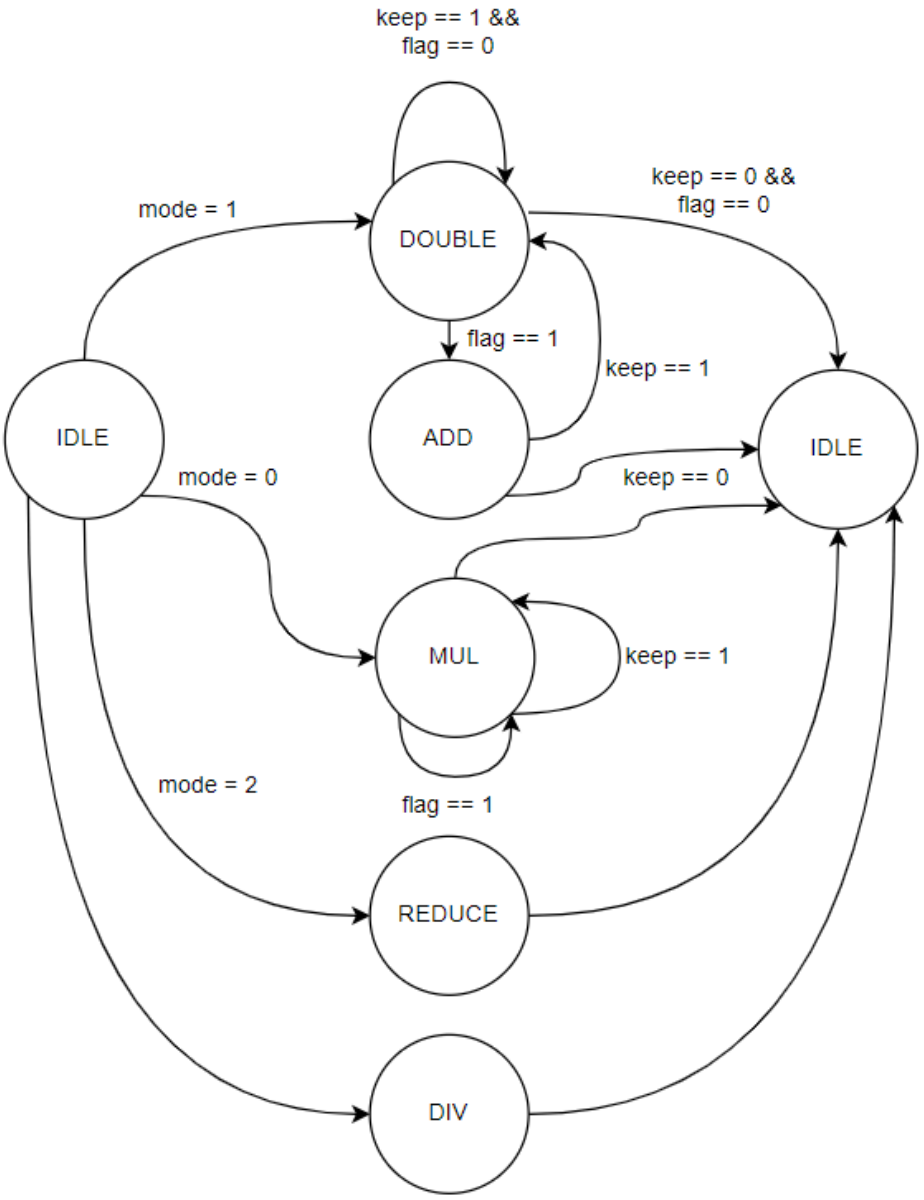

Figure 11. ALU FSM

Synthesis Results

Starting from the baseline design, we continuously improved our implementation. However, detailed APR was performed only for the final few design versions. Instead, we synthesized all intermediate design versions to compare their performance. **Table 12** presents the post-synthesis performance of all design versions. The specific differences between each design version are discussed in detail in this section.

Our baseline design uses the double-and-add algorithm for scalar multiplication and the TA-provided algorithm for modular inversion. Details of the modules architecture can be found in the **Hardware Implementation - Design Architecture** section. Except for modular multiplication and modular inversion, the modular addition and modular subtraction modules remain unchanged in the other design versions. The majority of our improvements focus on algorithm optimization rather than changes to the hardware architecture.

Version 2 design is an improvement based on the baseline design, focusing on optimizing the ALU FSM during scalar multiplication. Specifically, the IDLE state between the DOUBLE and ADD states was removed. For example, the baseline FSM sequence was: DOUBLE → IDLE → ADD → IDLE → DOUBLE …, while the updated sequence is: DOUBLE → ADD → DOUBLE …. This change saves 523 cycles and also reduces the synthesis area slightly. Overall, Version 2 achieves a **10.3%** performance improvement compared to the baseline design.

Version 3 is an optimization based on the Version 2 design. The data flow for point addition was rescheduled, reducing the calculation time from 10 cycles to 9 cycles, as shown previously in **Table 7**. This adjustment saves 123 cycles overall. Additionally, the adder and subtractor used in the REDUCE stage were merged into the ALU. Previously, separate hardware was used for these operations in the REDUCE stage; now, the ALU shares the hardware resources to perform these calculations. This change reduces the total area of the design. With these improvements, the overall AT performance has improved by **4.2%**.

Version 4 continues to optimize the calculation schedule. Specifically, we observed that the first cycle of point doubling calculates $Z \times Z$ for later use. However, this element can be precomputed during the last cycle of point addition or point doubling, provided it is not the first point doubling operation in scalar multiplication. By leveraging this optimization, we saved 254 cycles overall. Consequently, this improvement enhances the AT performance by **12.4%**.

In Version 5, we implemented a new modular inversion algorithm, as previously discussed, which reduced the cycle count by 505 compared to the Version 4 design. Additionally, we optimized the scheduling for point addition and point doubling to further reduce calculation cycles (schedules are shown in **Appendix**). Specifically, we observed that the T register is only required during point addition and not during point doubling. Thus, if the next state after

DOUBLE is also DOUBLE, the calculation of the T register can be skipped. Conversely, if the next state after DOUBLE is ADD, the T register must be calculated. Similarly, in the ADD state, the T register calculation can be skipped because the subsequent state must be DOUBLE. By combining these techniques, we achieved a savings of up to 761 cycles in Version 5 compared to Version 4 in PAT0. Although this optimization slightly increased the design area, it resulted in a 15.6% improvement in AT performance. Last, this marked the final optimization focused on algorithm improvements, after which our efforts shifted towards exploring alternative hardware architectures for further performance enhancements.

Version 6 represents the best design among all our implementations. It utilizes a 3-stage pipeline in the modular multiplication module, compared to the 2-stage pipeline used in previous designs. While increasing the pipeline stages results in a higher cycle count, it significantly reduces the critical path and, correspondingly, the clock period. Surprisingly, the area does not increase substantially, despite the additional registers required for the pipeline stages. Taking these trade-offs into account, Version 6 achieves an **8.4%** improvement in AT performance compared to Version 5. However, since we could not be certain that Version 6 would maintain better AT performance than Version 5 after APR, we conducted APR for both designs. Further APR details are presented in the **APR Results** section.

Design	cycles (PAT0)	clock period (ns)	area (um^2)	AxT (10^9)
Baseline	5346	10	1642733	87.82
Ver 2.	4823	10	1632199	78.72
Ver 3.	4700	10	1603902	75.38
Ver 4.	4446	9	1650036	66.02
Ver 5.	3685	9	1679940	55.71
Ver 6.	4444	7	1639938	51.02

Table 12. Synthesis results of different version designs

APR Results

The APR results for Version 5 and Version 6 are detailed in **Table 13**. Although Version 6 showed better performance during the synthesis stage, its post-simulation required an increased clock period of 8 ns to pass the tests. We suspect that the post-simulation violations were caused by the high core utilization rate during the APR stage, which made placement and routing more challenging for the tool.

Typically, negative slack issues are addressed using ECO techniques, but we could not ensure that repeated ECO adjustments would yield stable APR results. Our analysis indicated that a core utilization rate of 0.85 resulted in significant violations during post-simulation. Reducing the utilization rate to 0.82 allowed some patterns to pass, but others still failed. Ultimately, the issue could only be resolved by increasing the clock period.

In contrast, Version 5, with a core utilization rate of 0.85, successfully passed the tests. This suggests that the results may vary depending on the specific design and conditions, making them case-dependent.

This adjustment of time period negatively impacted Version 6's final AT score, making it less favorable overall. Consequently, we selected Version 5 as our final design. All APR information discussed in this section pertains to the Version 5 design.

Design	cycles (PAT0)	clock period (ns)	die area (um^2)	AxT (10^9)
Ver 5.	3685	9	2023576	67.11
Ver 6.	4444	8	2068403	73.53

Table 13. APR results of version 5 and Version 6

A. Layout

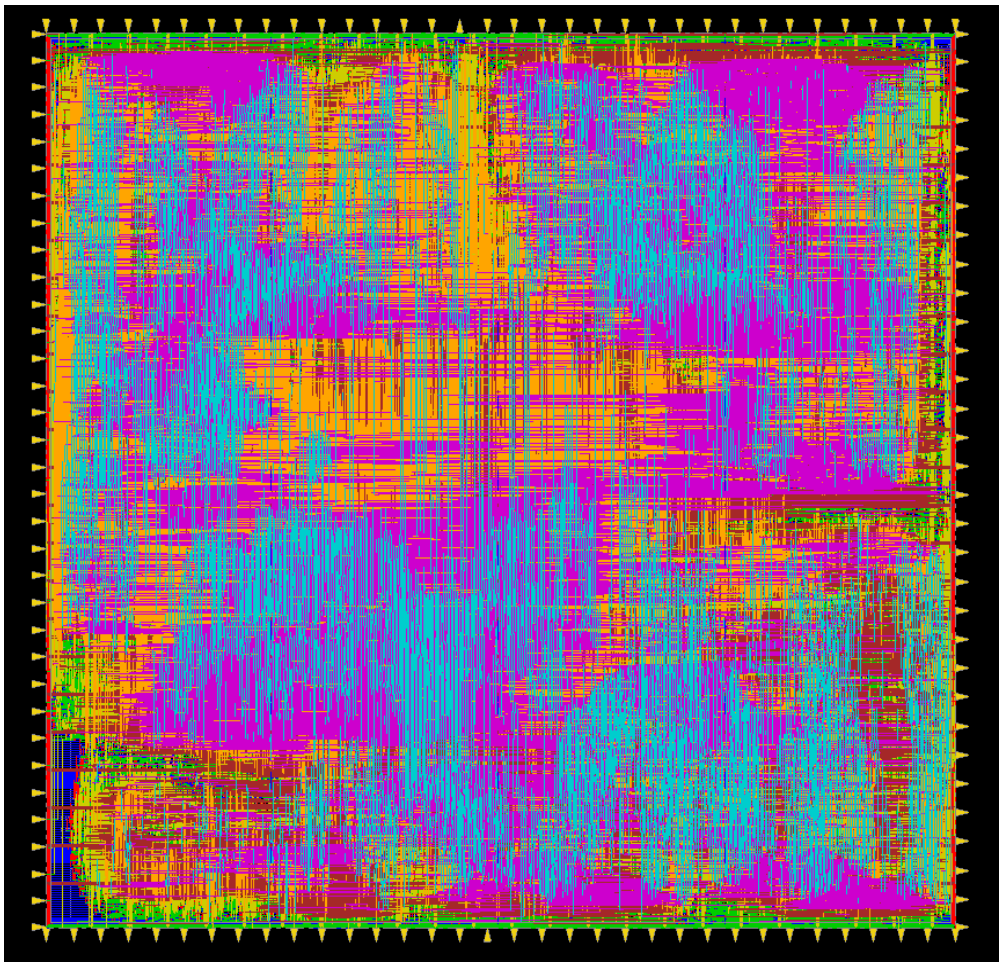

Figure 12. Layout of the design

B. LVS/DRC/Area results

```
innovus 1> verify_drc
 *** Starting Verify DRC (MEM: 1741.3) ***

  VERIFY DRC ..... Starting Verification
  VERIFY DRC ..... Initializing
  VERIFY DRC ..... Deleting Existing Violations
  VERIFY DRC ..... Creating Sub-Areas
  VERIFY DRC ..... Using new threading
  VERIFY DRC ..... Sub-Area: {0.000 0.000 179.520 179.520} 1 of 64
  VERIFY DRC ..... Sub-Area : 1 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {179.520 0.000 359.040 179.520} 2 of 64
  VERIFY DRC ..... Sub-Area : 2 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {359.040 0.000 538.560 179.520} 3 of 64
  VERIFY DRC ..... Sub-Area : 3 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {538.560 0.000 718.080 179.520} 4 of 64
  VERIFY DRC ..... Sub-Area : 4 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {718.080 0.000 897.600 179.520} 5 of 64
  VERIFY DRC ..... Sub-Area : 5 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {897.600 0.000 1077.120 179.520} 6 of 64
  VERIFY DRC ..... Sub-Area : 6 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {1077.120 0.000 1256.640 179.520} 7 of 64
  VERIFY DRC ..... Sub-Area : 7 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {1256.640 0.000 1426.460 179.520} 8 of 64
  VERIFY DRC ..... Sub-Area : 8 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {0.000 179.520 179.520 359.040} 9 of 64
  VERIFY DRC ..... Sub-Area : 9 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {179.520 179.520 359.040 359.040} 10 of 64
  VERIFY DRC ..... Sub-Area : 10 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {359.040 179.520 538.560 359.040} 11 of 64
  VERIFY DRC ..... Sub-Area : 11 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {538.560 179.520 718.080 359.040} 12 of 64
  VERIFY DRC ..... Sub-Area : 12 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {718.080 179.520 897.600 359.040} 13 of 64
  VERIFY DRC ..... Sub-Area : 13 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {897.600 179.520 1077.120 359.040} 14 of 64
  VERIFY DRC ..... Sub-Area : 14 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {1077.120 179.520 1256.640 359.040} 15 of 64
```

~~~~~~

```
  VERIFY DRC ..... Sub-Area: {718.080 1256.640 897.600 1418.600} 61 of 64
  VERIFY DRC ..... Sub-Area : 61 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {897.600 1256.640 1077.120 1418.600} 62 of 64
  VERIFY DRC ..... Sub-Area : 62 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {1077.120 1256.640 1256.640 1418.600} 63 of 64
  VERIFY DRC ..... Sub-Area : 63 complete 0 Viols.
  VERIFY DRC ..... Sub-Area: {1256.640 1256.640 1426.460 1418.600} 64 of 64
  VERIFY DRC ..... Sub-Area : 64 complete 0 Viols.

  Verification Complete : 0 Viols.

*** End Verify DRC (CPU: 0:01:12  ELAPSED TIME: 124.00  MEM: 0.0M) ***
```

Figure 13. DRC check result
~~~~~~

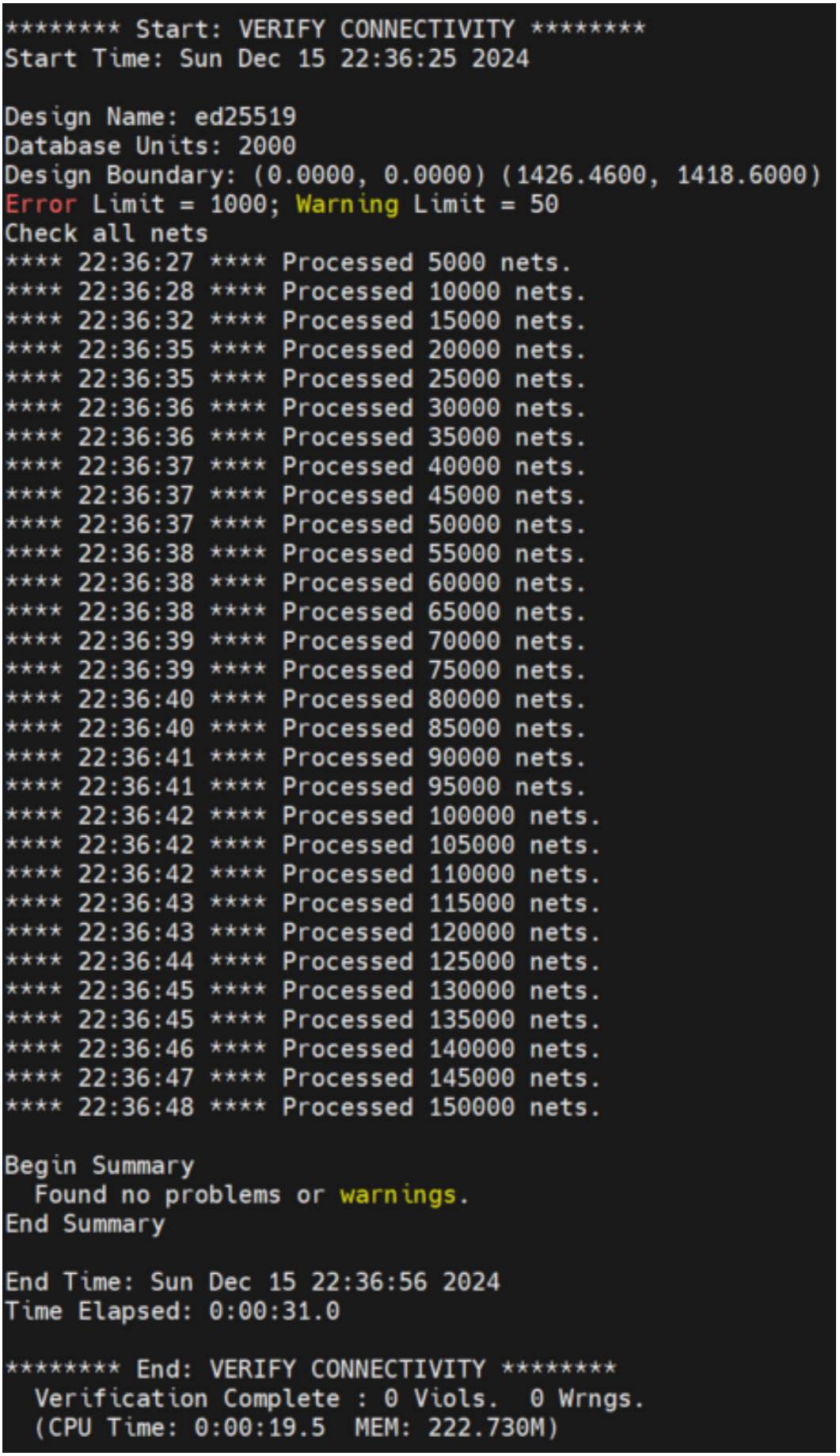

Figure 14. LVS check result

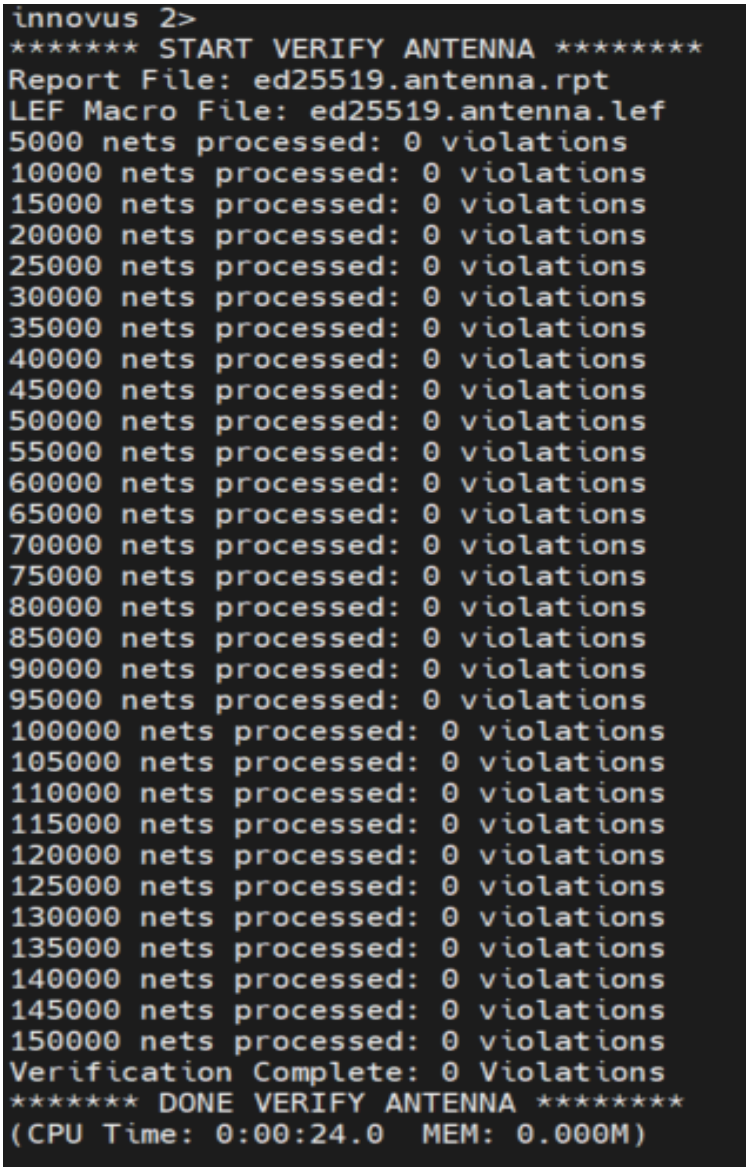

Figure 15. Antenna check result

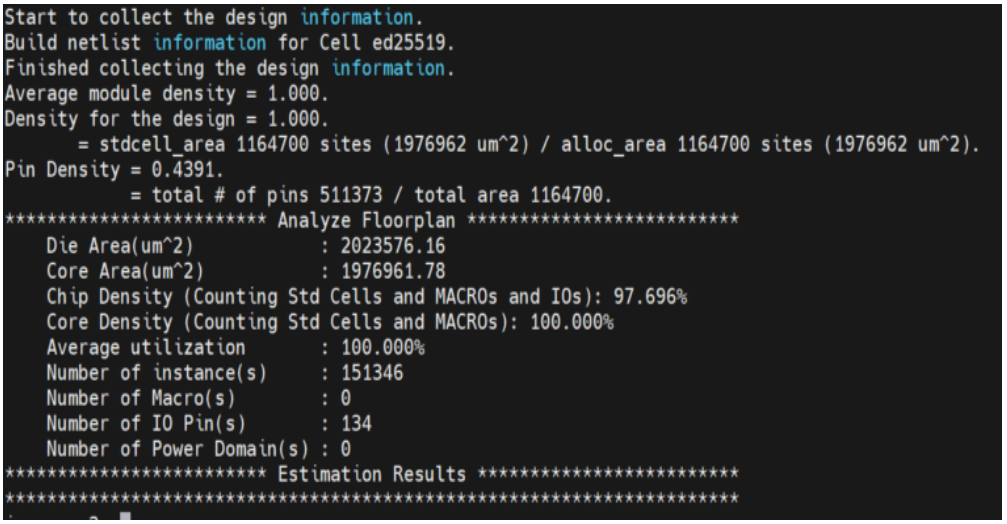

Figure 16. Area information

Reference

[1] Edwards-curve Digital Signature Algorithm - https://zh.wikipedia.org/zh-tw/EdDSA
[2] Twisted Edwards curve coordinate: https://en.wikipedia.org/wiki/Twisted_Edwards_curve
[3] 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
[4] Yiming XUE, Shurong LIU, Shuheng GUO, Yan LI, Cai'e HU. High-speed hardware architecture design and implementation of Ed25519 signature verification algorithm[J]. Journal on Communications, 2022, 43(3): 101-112.
[5] NTU CVSD2024 Final Project description file

Appendix

Algorithm

- **scalar multiplication: double-and-add [5]**

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

- **TA-provided modular inversion [5]**

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

Special Schedule for Point Addition and Point Doubling in Version 5 design

- **Point Doubling**

Cycle	Mul	ModQ	ADD	SUB
(0)	$t_1 = Z_1 * Z_1$			
(1)	$A = X_1 * X_1$	t_1	$t_2 = X_1 + Y_1$	
(2)	$B = Y_1 * Y_1$	A	$C = t_1 + t_1$	
(3)	$t_2 = t_2 * t_2$	B		
(4)		t_2	$H = A + B$	$G = A - B$
(5)	$Y_3 = G * H$		$F = C + G$	$E = H - t_2$
(6)	$Z_3 = F * G$	Y_3		
(7)	$X_3 = F * G$	Z_3		
(8)	$T_3 = F * G$	X_3		
(9)	$t_1 = Z_1 * Z_1$	T_3		

- **Point Addition**

Cycle	Mul	ModQ	ADD	SUB
(0)	$C = T_1 * T_2$		$B_1 = X_1 + Y_1$	$A_1 = Y_1 - X_1$
(1)	$A = A_1 * A_2$	C	$t_2 = Z_1 + Z_1$	
(2)	$B = B_1 * B_2$	A	$C = C + C$	
(3)		B	$G = t_2 + C$	$F = t_2 - C$
(4)	$Z_3 = F * G$		$H = A + B$	$G = B - A$
(5)	$Y_3 = G * H$	Z_3		
(6)	$X_3 = E * F$	Y_3		
(7)	$t_1 = Z_1 * Z_1$	X_3		

Test Patterns

- **Test Pattern 1 (PAT0)**

  M: 0x259f4329e6f4590b9a164106cf6a659eb4862b21fb97d43588561712e8e5216a
  X: 0x0fa4d2a95dafe3275eaf3ba907dbb1da819aba3927450d7399a270ce660d2fae
  Y: 0x2f0fe2678dedf6671e055f1a557233b324f44fb8be4afe607e5541eb11b0bea2

- **Test Pattern 2 (PAT1)**

  M: 0x17e0aa3c03983ca8ea7e9d498c778ea6eb2083e6ce164dba0ff18e0242af9fc3
  X: 0x2e2c9fbf00b87ab7cde15119d1c5b09aa9743b5c6fb96ec59dbf2f30209b133c
  Y: 0x116943db82ba4a31f240994b14a091fb55cc6edd19658a06d5f4c5805730c232

- **Test Pattern 3 (PAT2)**

  M: 0x1759edc372ae22448b0163c1cd9d2b7d247a8333f7b0b7d2cda8056c3d15eef7
  X: 0x5b90ea17eaf962ef96588677a54b09c016ad982c842efa107c078796f88449a8
  Y: 0x6a210d43f514ec3c7a8e677567ad835b5c2e4bc5dd3480e135708e41b42c0ac6

ALU Architecture