CVSD Final Project
Elliptic Curve Cryptographic Processor
Team 12

1. APR Results

We used the following specifications to perform P&R:

Core Size Ratio	Core Utilization	Core Margin
1	0.875	8

Table 1. Floorplan Specification

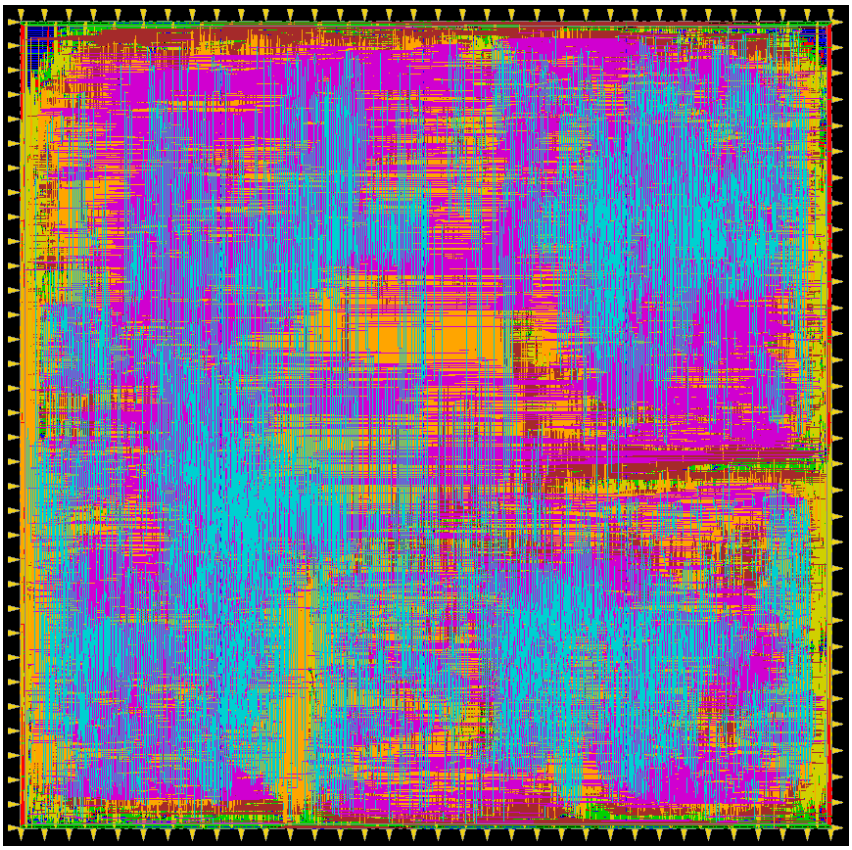
Ring Width	Number of Bits
2	1

Table 2. Power Rings Specification

Stripe Width	Set-to-Set Distance
2	50

Table 3. Power Stripes Specification (Vertical Stripe used only)

(1) Layout



(2) DRC Check

```
 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: {1191.360 0.000 1385.520 198.560} 7 of 49  Thread : 0
VERIFY DRC ..... Sub-Area: {0.000 0.000 198.560 198.560} 1 of 49  Thread : 1
VERIFY DRC ..... Sub-Area: {794.240 0.000 992.800 198.560} 5 of 49  Thread : 4
VERIFY DRC ..... Sub-Area: {0.000 397.120 198.560 595.680} 15 of 49  Thread : 5
VERIFY DRC ..... Sub-Area: {397.120 0.000 595.680 198.560} 3 of 49  Thread : 2
VERIFY DRC ..... Sub-Area: {1191.360 397.120 1385.520 595.680} 21 of 49  Thread : 7
VERIFY DRC ..... Sub-Area: {397.120 397.120 595.680 595.680} 17 of 49  Thread : 3
VERIFY DRC ..... Sub-Area: {794.240 397.120 992.800 595.680} 19 of 49  Thread : 6
VERIFY DRC ..... Sub-Area: {0.000 794.240 198.560 992.800} 29 of 49  Thread : 0
VERIFY DRC ..... Sub-Area: {1191.360 595.680 1385.520 794.240} 28 of 49  Thread : 7
VERIFY DRC ..... Sub-Area: {397.120 794.240 595.680 992.800} 31 of 49  Thread : 1
VERIFY DRC ..... Sub-Area: {595.680 397.120 794.240 595.680} 18 of 49  Thread : 6
VERIFY DRC ..... Sub-Area: {0.000 198.560 198.560 397.120} 8 of 49  Thread : 5
VERIFY DRC ..... Sub-Area: {595.680 0.000 794.240 198.560} 4 of 49  Thread : 2
VERIFY DRC ..... Sub-Area: {794.240 794.240 992.800 992.800} 33 of 49  Thread : 3
VERIFY DRC ..... Sub-Area: {992.800 0.000 1191.360 198.560} 6 of 49  Thread : 4
VERIFY DRC ..... Sub-Area: {0.000 595.680 198.560 794.240} 22 of 49  Thread : 0
VERIFY DRC ..... Sub-Area: {992.800 397.120 1191.360 595.680} 20 of 49  Thread : 7
VERIFY DRC ..... Sub-Area: {198.560 0.000 397.120 198.560} 2 of 49  Thread : 5
VERIFY DRC ..... Sub-Area: {1191.360 794.240 1385.520 992.800} 35 of 49  Thread : 1
VERIFY DRC ..... Sub-Area: {397.120 595.680 595.680 794.240} 24 of 49  Thread : 6
VERIFY DRC ..... Sub-Area: {595.680 992.800 794.240 1191.360} 39 of 49  Thread : 4
VERIFY DRC ..... Sub-Area: {992.800 1191.360 1191.360 1378.010} 48 of 49  Thread : 0
VERIFY DRC ..... Sub-Area: {595.680 198.560 794.240 397.120} 11 of 49  Thread : 2
VERIFY DRC ..... Sub-Area: {198.560 992.800 397.120 1191.360} 37 of 49  Thread : 3
VERIFY DRC ..... Sub-Area: {992.800 198.560 1191.360 397.120} 13 of 49  Thread : 7
VERIFY DRC ..... Sub-Area: {198.560 198.560 397.120 397.120} 9 of 49  Thread : 5
VERIFY DRC ..... Sub-Area: {794.240 595.680 992.800 794.240} 26 of 49  Thread : 1
VERIFY DRC ..... Sub-Area: {0.000 992.800 198.560 1191.360} 36 of 49  Thread : 3
VERIFY DRC ..... Sub-Area: {992.800 992.800 1191.360 1191.360} 41 of 49  Thread : 0
VERIFY DRC ..... Sub-Area: {595.680 1191.360 794.240 1378.010} 46 of 49  Thread : 4
VERIFY DRC ..... Sub-Area: {794.240 198.560 992.800 397.120} 12 of 49  Thread : 7
VERIFY DRC ..... Sub-Area: {397.120 198.560 595.680 397.120} 10 of 49  Thread : 5
VERIFY DRC ..... Sub-Area: {1191.360 198.560 1385.520 397.120} 14 of 49  Thread : 2
VERIFY DRC ..... Sub-Area: {198.560 595.680 397.120 794.240} 23 of 49  Thread : 6
VERIFY DRC ..... Sub-Area: {595.680 595.680 794.240 794.240} 25 of 49  Thread : 1
VERIFY DRC ..... Sub-Area: {992.800 595.680 1191.360 794.240} 27 of 49  Thread : 3
VERIFY DRC ..... Sub-Area: {1191.360 992.800 1385.520 1191.360} 42 of 49  Thread : 0
VERIFY DRC ..... Sub-Area: {794.240 992.800 992.800 1191.360} 40 of 49  Thread : 7
VERIFY DRC ..... Sub-Area: {397.120 992.800 595.680 1191.360} 38 of 49  Thread : 4
VERIFY DRC ..... Sub-Area: {198.560 397.120 397.120 595.680} 16 of 49  Thread : 5
VERIFY DRC ..... Sub-Area: {0.000 1191.360 198.560 1378.010} 43 of 49  Thread : 0
VERIFY DRC ..... Sub-Area: {1191.360 1191.360 1385.520 1378.010} 49 of 49  Thread : 1
VERIFY DRC ..... Sub-Area: {992.800 794.240 1191.360 992.800} 34 of 49  Thread : 7
VERIFY DRC ..... Sub-Area: {794.240 1191.360 992.800 1378.010} 47 of 49  Thread : 2
VERIFY DRC ..... Sub-Area: {397.120 1191.360 595.680 1378.010} 45 of 49  Thread : 3
VERIFY DRC ..... Thread : 2 finished.
VERIFY DRC ..... Sub-Area: {198.560 794.240 397.120 992.800} 30 of 49  Thread : 4
VERIFY DRC ..... Thread : 4 finished.
VERIFY DRC ..... Sub-Area: {595.680 794.240 794.240 992.800} 32 of 49  Thread : 6
VERIFY DRC ..... Thread : 6 finished.
VERIFY DRC ..... Sub-Area: {198.560 1191.360 397.120 1378.010} 44 of 49  Thread : 3
VERIFY DRC ..... Thread : 3 finished.

 Verification Complete : 0 Viols.
```

(3) LVS Check

```
Design Name: ed25519
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (1385.5200, 1378.0100)
Error Limit = 1000; Warning Limit = 50
Check all nets
Use 8 pthreads

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Sat Dec 14 13:18:23 2024
Time Elapsed: 0:00:07.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:22.8  MEM: 0.000M)
```

(4) Process Antenna Check

```
Report File: ed25519.antenna.rpt
LEF Macro File: ed25519.antenna.lef
5000 nets processed: 0 violations
10000 nets processed: 0 violations
15000 nets processed: 0 violations
20000 nets processed: 0 violations
25000 nets processed: 0 violations
30000 nets processed: 0 violations
35000 nets processed: 0 violations
40000 nets processed: 0 violations
45000 nets processed: 0 violations
50000 nets processed: 0 violations
55000 nets processed: 0 violations
60000 nets processed: 0 violations
65000 nets processed: 0 violations
70000 nets processed: 0 violations
75000 nets processed: 0 violations
80000 nets processed: 0 violations
85000 nets processed: 0 violations
90000 nets processed: 0 violations
95000 nets processed: 0 violations
100000 nets processed: 0 violations
105000 nets processed: 0 violations
110000 nets processed: 0 violations
115000 nets processed: 0 violations
120000 nets processed: 0 violations
125000 nets processed: 0 violations
130000 nets processed: 0 violations
135000 nets processed: 0 violations
140000 nets processed: 0 violations
145000 nets processed: 0 violations
Verification Complete: 0 Violations
******* DONE VERIFY ANTENNA ********
(CPU Time: 0:00:23.1  MEM: 0.000M)
```

(5) Setup Time Check

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 av_func_mode_max

+--------------------+---------+---------+---------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | in2reg  | reg2out | in2out  | default |
+--------------------+---------+---------+---------+---------+---------+---------+
|           WNS (ns):|  0.126  |  0.126  |  3.767  |  3.538  |   N/A   |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |  0.000  |   N/A   |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |    0    |   N/A   |    0    |
|          All Paths:|  3936   |  3870   |  3870   |   66    |   N/A   |    0    |
+--------------------+---------+---------+---------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 94.834%
       (100.000% with Fillers)
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 194.59 sec
Total Real time: 106.0 sec
Total Memory Usage: 2659.878906 Mbytes
Reset AAE Options
```

(6) Hold Time Check

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Hold  views included:
 av_func_mode_max

+--------------------+---------+---------+---------+---------+---------+---------+
|     Hold mode      |   all   | reg2reg | in2reg  | reg2out | in2out  | default |
+--------------------+---------+---------+---------+---------+---------+---------+
|           WNS (ns):|  0.948  |  0.948  |  1.002  |  3.065  |   N/A   |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |  0.000  |   N/A   |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |    0    |   N/A   |    0    |
|          All Paths:|  3936   |  3870   |  3870   |   66    |   N/A   |    0    |
+--------------------+---------+---------+---------+---------+---------+---------+

Density: 94.834%
       (100.000% with Fillers)
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 184.94 sec
Total Real time: 99.0 sec
Total Memory Usage: 2567.757812 Mbytes
Reset AAE Options
```

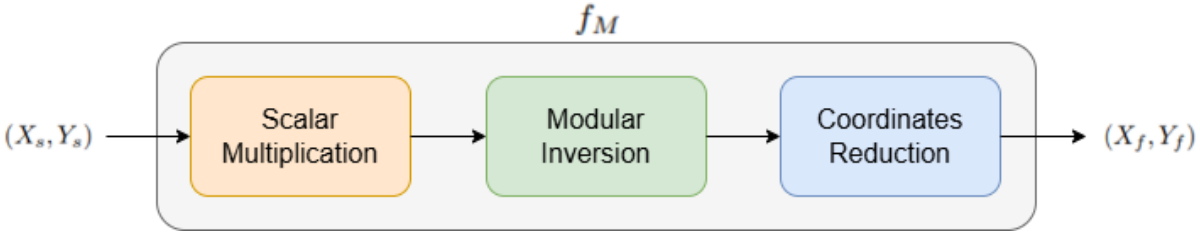
(7) Floorplan Analysis

```
Start to collect the design information.
Build netlist information for Cell ed25519.
Finished collecting the design information.
Average module density = 1.000.
Density for the design = 1.000.
       = stdcell_area 1098144 sites (1863990 um^2) / alloc_area 1098144 sites (1863990 um^2).
Pin Density = 0.4701.
            = total # of pins 516273 / total area 1098144.
************************* Analyze Floorplan *************************
    Die Area(um^2)             : 1909260.42
    Core Area(um^2)            : 1863989.63
    Chip Density (Counting Std Cells and MACROs and IOs): 97.629%
    Core Density (Counting Std Cells and MACROs): 100.000%
    Average utilization        : 100.000%
    Number of instance(s)      : 150519
    Number of Macro(s)         : 0
    Number of IO Pin(s)        : 134
    Number of Power Domain(s) : 0
************************* Estimation Results *************************
**********************************************************************
```

2. Algorithm Design

(1) System Model

In this project, we design the core function in EdDSA, which transforms an integer point to another on the twisted Edward curve.



Where (X_S, Y_S) is the input start point on the curve, f_M is the transformation depends on the input scalar M, and (X_f, Y_f) is the output destination point. The overall transformation process f_M can be divided into three parts: Scalar Multiplication, Modular Inversion, and Coordinates Reduction.

(a) Scalar Multiplication

We use a series of Point Double and Point Addition processes. The Point Double and Addition consists of several modular multiplications, additions, and subtractions.

The algorithm is described below:

Algorithm 1 Scalar Multiplication

Input: P: Point on the curve,
M: Binary representation of the scalar,
n: Bit length of the scalar
Output: Resultant point r
$r \leftarrow 0$
for i from $n-1$ to 0 **do**
  $r \leftarrow 2r$
  if ith bit of M is 1 **then**
    $r \leftarrow r + P$
  end if
end for
return r

(b) Modular Inversion

We apply Fermat's little theorem to find the inverse element.

$$Z^{-1} mod q = Z^{q-2} mod q$$

This involves a series of modular multiplications. However, to implement a more efficient multiplication process in hardware perspective, we use the below algorithm to do Modular Inversion:

Algorithm 2 Modular Inversion

Input: Z: Divisor
Output: $Z^{-1} mod q$
Forward Multiplication
$r \leftarrow 1$
$\hat{Z} \leftarrow Z$
for i from 1 to 249 **do**
    $r \leftarrow r \cdot \hat{Z}$
    $\hat{Z} \leftarrow \hat{Z} \cdot \hat{Z}$
end for
Self Multiplication
for i from 1 to 5 **do**
    $r \leftarrow r \cdot r$
end for
Post Multiplication
$s \leftarrow Z^3$
$t \leftarrow Z^8$
$\dot{Z} \leftarrow s \cdot t$
$r \leftarrow r \cdot \dot{Z}$
$Z^{2^{250}-1}$
$Z^{2^{255}-32}$
$Z^{2^{255}-21}$
return r

The algorithm gives us $Z^{q-2} mod q$ at the end. The efficiency of this algorithm will be further discussed in the later section.

(c) Coordinates Reduction:

We can use the below algorithm to find the output at normal coordinates after finding the inverse element of Z. It is achieved by a series of modular multiplications and scalar subtraction.

The algorithm is described below:

Algorithm 3 Coordinates Reduction

Input: Z: Divisor,
X: Initial x-coordinate,
Y: Initial y-coordinate,
q: Modular base
Output: Adjusted coordinates (x, y)
$inv \leftarrow Mod\_Inv(Z)$ # Output of Modular Inversion
$x \leftarrow X \cdot inv$
$y \leftarrow Y \cdot inv$
if x is odd **then**
    $x \leftarrow q - x$
end if
if y is odd **then**
    $y \leftarrow q - y$
end if
return (x, y)

(d) Analyze the number of operations

We can analyze the number of modular operations for these three types of computations.

# Modular Operations	Scalar Multiplication	Modular Inversion	Coordinates Reduce
Multiplication	8095	505	2
Addition / Subtraction	2310	0	2 (Scalar)

Table 4. Number of Modular Operations for each part

Note that the number is based on projective coordinates implementation.

(2) Optimization of Point Double and Point Addition

The previous part shows that Scalar Multiplication dominates the number of modular operations. Therefore, we first optimize the efficiency of Point Double and Addition. To achieve this, we use Extended Coordinate Representation instead of Projective Coordinates.

The Point Double process can be formulated as below:

Algorithm 4 Point Double

Input: (X_1, Y_1, Z_1, T_1): Point 1 on the curve,
(X_2, Y_2, Z_2, T_2): Point 2 on the curve
Output: (X_3, Y_3, Z_3, T_3): Resultant point after addition
$t1 \leftarrow Z_1 \cdot Z_1$
$A \leftarrow X_1 \cdot X_1$
$t2 \leftarrow X_1 + Y_1$
$B \leftarrow Y_1 \cdot Y_1$
$C \leftarrow t1 + t1$
$t2 \leftarrow t2 \cdot t2$
$t2 \leftarrow t2 - A$
$t2 \leftarrow t2 - B$
$H \leftarrow A - B$
$G \leftarrow C - B$
$Y_3 \leftarrow G \cdot H$
$F \leftarrow C + B$
$E \leftarrow H - t2$
$Z_3 \leftarrow H \cdot G$
$T_3 \leftarrow E \cdot H$
$X_3 \leftarrow E \cdot F$
return (X_3, Y_3, Z_3, T_3)

The Point Addition process can be formulated as below:

Algorithm 5 Point Addition

Input: (X_1, Y_1, Z_1, T_1): Point 1 on the curve,
(X_2, Y_2, Z_2, T_2): Point 2 on the curve
Output: (X_3, Y_3, Z_3, T_3): Resultant point after addition
$A2 \leftarrow Y_2 - X_2$
$B2 \leftarrow Y_2 + X_2$
$T2 \leftarrow T_1 \cdot T_2$
$C \leftarrow Z_1 \cdot Z_1$
$t2 \leftarrow Z_1 \cdot Z_2$
$D \leftarrow t2 \cdot Z_2$
$B1 \leftarrow Y_1 + X_1$
$A1 \leftarrow Y_1 - X_1$
$A \leftarrow A1 \cdot A2$
$C \leftarrow C + C$
$B \leftarrow B1 \cdot B2$
$G \leftarrow D - C$
$F \leftarrow D + C$
$Z_3 \leftarrow H \cdot G$
$H \leftarrow B - A$
$E \leftarrow B + A$
$Y_3 \leftarrow G \cdot H$
$X_3 \leftarrow E \cdot F$
$T_3 \leftarrow E \cdot H$
return (X_3, Y_3, Z_3, T_3)

We then compare the number of modular operations needed for Extended Coordinates and Projective Coordinates:

	Projective Coordinates	Extended Coordinates
Multiplication	21	8
Addition / Subtraction	6	6

Table 5. Number of Modular Operations for Point Double

	Projective Coordinates	Extended Coordinates
Multiplication	21	9
Addition / Subtraction	6	10

Table 6. Number of Modular Operations for Point Addition

We observe that using Extended Coordinates Representation significantly reduces the number of modular multiplications. Although the number of modular addition and subtraction slightly increases in Point Addition, these additional computations won't affect the overall throughput of our hardware. We will have further discussion in the later section.

(3) Optimization of Modular Multiplication

Instead of using Montgomery Multiplication (MM), we simply implement modular multiplication as below:

Algorithm 6 Simplified Modular Multiplication

1: **Input:** x: Multiplicand 1,
y: Multiplicand 2,
q: Prime number
2: **Output:** Result of modular multiplication r
3: $m \leftarrow x \cdot y$
4: $r \leftarrow m\%q$
5: **return** r

It consists of a multiplication between two 255-bit numbers and a modulo operation.

We first use the Karatsuba Algorithm to split it into 128x128 multiplications and additions, which are described as below:

Karatsuba Algorithm

$$H \leftarrow x[255:128] \cdot y[255:128]$$
$$L \leftarrow x[127:0] \cdot y[127:0]$$
$$M \leftarrow \big(x[255:128] + x[127:0]\big) \cdot \big(y[255:128] + y[127:0]\big) - H - L$$
$$m \leftarrow (H << 256) + (M << 128) + L$$

Note that m will be the product of the original multiplication.

Since q is a constant and $2^{256} \pmod q = 38$, we can simplify the equation of m as T:

Simplification Stage 1

$$Ch \leftarrow 38 \cdot H + L$$
$$Cl \leftarrow (M \ll 128)$$
$$T \leftarrow Ch + Cl$$

Moreover, since T is at most 389-bit and 2^{256} (mod q) = 19, we can further simplify T as T2:

Simplification Stage 2
$Th \leftarrow T[388:255]$
$Tl \leftarrow T[254:0]$
$T2 \leftarrow 19 \cdot Th + Tl$

Finally, since T2 is at most 256-bit, we can perform at most 1 subtraction to get the result of modular multiplication:
The overall algorithm of our modular multiplication can be formulated as:

Simplification Stage 3
if $T2 > q$ **then**
  $r \leftarrow T2 - q$
else
  $r \leftarrow T2$
end if

The overall algorithm of our modular multiplication can be formulated as:

Algorithm 7 Modular Multiplication

Input: x: Binary representation of Multiplicand 1,
y: Binary representation of Multiplicand 2,
q: Prime number
Output: Result of modular multiplication r
Karatsuba Algorithm
$H \leftarrow x[255:128] \cdot y[255:128]$
$L \leftarrow x[127:0] \cdot y[127:0]$
$M \leftarrow \big(x[255:128] + x[127:0]\big) \cdot \big(y[255:128] + y[127:0]\big) - H - L$
Simplification Stage 1
$Ch \leftarrow 38 \cdot H + L$
$Cl \leftarrow (M \ll 128)$
$T \leftarrow Ch + Cl$
Simplification Stage 2
$Th \leftarrow T[388:255]$
$Tl \leftarrow T[254:0]$
$T2 \leftarrow 19 \cdot Th + Tl$
Simplification Stage 3
if $T2 > q$ **then**
  $r \leftarrow T2 - q$
else
  $r \leftarrow T2$
end if
return r

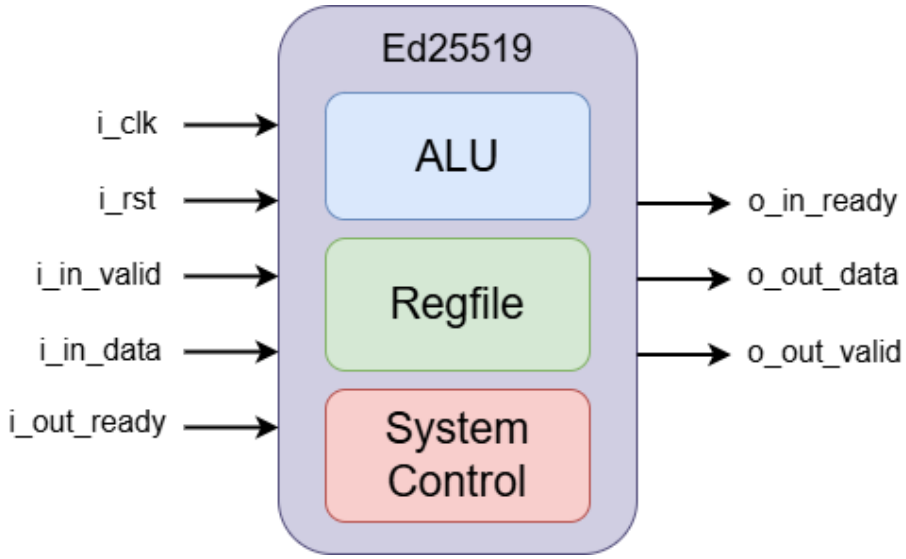
The Montgomery Multiplication (MM) needs six 255x255 multiplications to complete a modular multiplication. On the other hand, our implementation only needs three 128x128 multiplications, which is much more efficient and friendly for hardware implementation.

	MM	Ours
# Multiplication	6	3
Bit-width	255x255	128x128

Table 7. Number and Bit-width needed for a Modular Multiplication

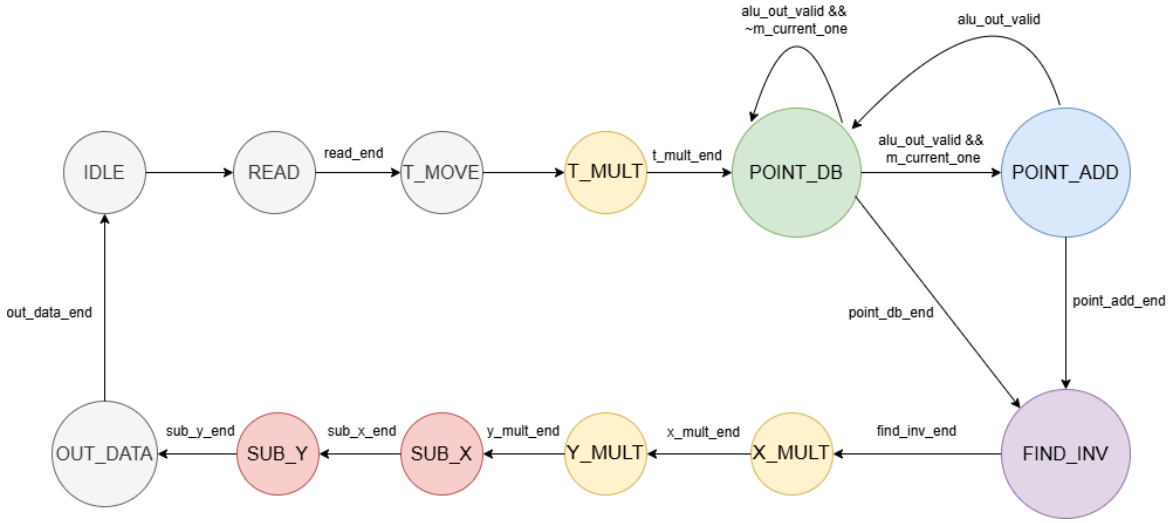
3. Hardware Implementation

Our hardware implementation can be divided into three parts: System Control, ALU, and Regfile:



(1) System Control

System Control manage the entire function flow of our hardware. Since we use Double and Add algorithm to implement Scalar Multiplication, Fermat's little theorem to find Modular Inverse and perform Coordinates Reduction at the end, we can design the System Finite State Machine as below:



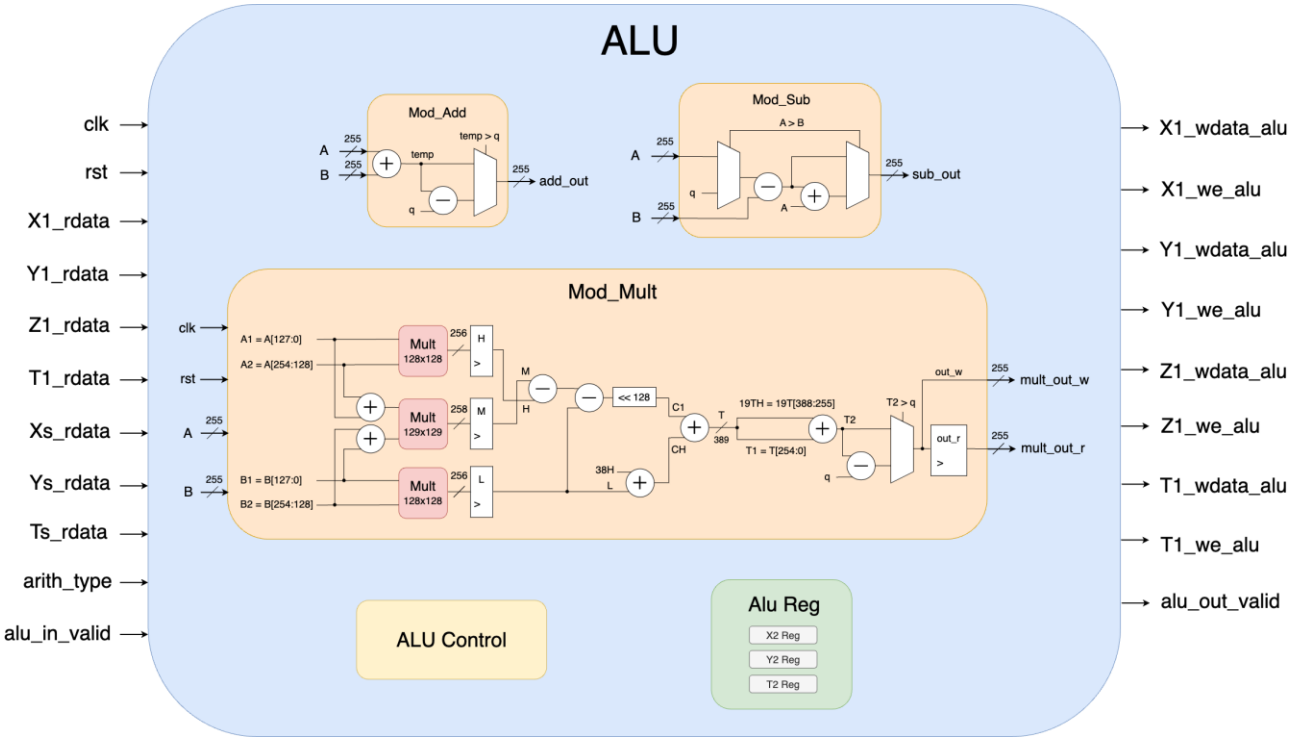
The main function of each state is described below:

READ: Read input data from the testbed
T_MOVE: Move X, Y of zero point to register T1 and register Ts respectively
T_MULT: Multiply X, Y to get T of zero point
POINT_DB: Perform Point Double in Scalar Multiplication
POINT_ADD: Perform Point Addition in Scalar Multiplication
FIND_INV: Perform Modular Inversion
X_MULT: x = X/Z in Coordinate Reduction
Y_MULT: y = Y/Z in Coordinate Reduction
SUB_X: q-x if x is odd
SUB_Y: q-y if y is odd
OUT_DATA: Write output data to testbed

During each state, System Control may send request (alu_in_valid) with type of operation (arith_type) to ALU and wait for ack (alu_out_valid) to complete a computation, or directly read / write the content in Regfile.

(2) ALU

ALU Module perform various types of arithmetic operation. It contains ALU Control, ALU Reg, a modular adder, a modular subtractor, and a modular multiplier below:



In Mod_Mult, three registers are used to store 130x130 multiplication outputs and one is used to store modulo output. Also, in ALU Reg, three registers (X2, Y2, T2) are used to store pre-calculated LUTs. After ALU receives the alu_in_valid from System Control, it performs five types of arithmetic operations. The input data will be read from registers in Regfile, and output data will be written back to it. After the operation is completed, ALU will raise alu_out_valid to high and System Control can check data in Regfile. Our ALU supports four types of operations described below:

Point_DB: Point Double
Point_ADD: Point Addition
Num_Mult: Modular Multiplication
Scalr_Sub: Scalar Subtraction
Mod_Inv: Find Modular Inverse

Our ALU determines the type of the next arithmetic operation at the last cycle of the current operation. This ALU arithmetic pre-fetch enable our ALU to directly start the next computation immediately when it receives the next alu_in_valid.

	arith_type	Input Register	Output Register	# Cycle
Point_DB	3'b000	X1, Y1, Z1, T1	X1, Y1, Z1, T1	9
Point_ADD	3'b001	X1, Y1, Z1, T1, Xs, Ys, Ts	X1, Y1, Z1, T1	9
Num_Mult	3'b010	T1, Ts	T1	2
Scalr_Sub	3'b011	T1, Ts	T1	1
Mod_Inv	3'b100	Z1	T1	516

Table 8. I/O Port and Total Cycles for ALU operations

Note that for each Point_DB and Point_ADD operation, we need to schedule the usage of our adder, subtractor, and multiplier instance. The scheduling of each cycle is listed below, LHS represents output register and RHS represents input register:

*Mo: multiplier output register

Cycle	Multiplier	Mod	Adder	Subtractor
0	Mo = Z1·Z1			
1	X1 = X1·X1	Mo	Z1 = X1 + Y1	
2	Mo = Y1·Y1	X1	T1 = Mo + Mo	
3	Mo = Z1·Z1	Mo		
4		Mo	Z1 = X1 + Mo	Y1 = X1 - Mo
5	Y1 = Y1·Z1		T1 = T1 + Y1	X1 = Z1 - Mo
6	Z1 = T1·Y1	Y1		
7	T1 = X1·Z1	Z1		
8	X1 = X1·T1	T1		
9	Mo = Z1·Z1	X1		

Table 9. Point Double Operation Schedule

Cycle	Multiplier	Mod	Adder	Subtractor
0	T2 = Ts·d		Y2 = Ys + Xs	X2 = Ys - Xs
1		T2		
2	Mo = T1·T2		Y1 = Y1 + X1	X1 =Y1 - X1
3	X1 = X1·X2	Mo	Z1 = Z1 + Z1	
4	Mo = Y1·Y2	X1	T1 = Mo + Mo	
5		Mo	Z1 = Z1 + T1	Y1 = Z1 - T1
6	Z1 = Y1·Z1			
7	Y1 = Z1·X1	Z1	X1 = Mo + X1	T1= Mo - X1
8	X1 = T1·Y1	Y1		
9	T1 = T1·X1	X1		
10	Mo = Z1·Z1	T1		

Table 10. Point Addition Operation Schedule

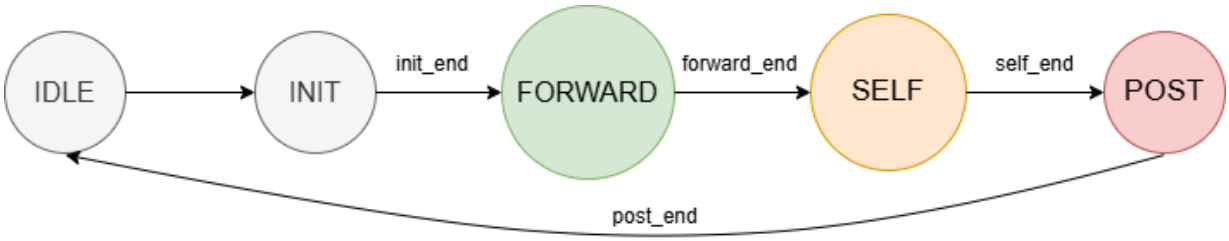
We also apply two techniques to further reduce the cycle required for each Point Double / Addition operation.

(a) We can compute Mo = Z1·Z1 at the last cycle of each Point Double / Addition operation. Therefore, we can directly start from cycle 1 after the first Point Double operation. The average cycle required for Point Double is 9.

(b) Since Xs, Ys, Zs is fixed in each pattern, LUTs (X2, Y2, T2) are constants and need to be computed only once. Therefore, after the first Point Addition operation, we can skip the pre-calculation and directly start from cycle 2. The average cycle required for Point Addition is 9.

We can also see that for Point Double / Addition, the decisive factor for cycle required is the total number of modular multiplications. This justifies that "additional modular addition and subtraction in Extended Coordinates Representation won't affect the overall throughput of our hardware".

To efficiently perform Modular Inversion, we use algorithm 2 in the previous part and design the following sub-FSM for ALU:



The main function of each state is described below:

INIT: Move Z, 1 to T1 register, T2 register respectively
FORWARD: Perform multiplication to get $Z^{2^{250}-1}$
SELF: Perform multiplication to get $Z^{2^{255}-32}$
POST: Perform multiplication to get $Z^{2^{255}-21}$

For FORWARD state, the schedule of multiplier instance is below:

Cycle	Multiplier	Modp	T1 Reg	T2 Reg
0	Z^2 = T1 · T1	-	-	-
1	Z = T1 · T2	Z^2	-	-
2	Z^4 = T1 · T1	Z	Z^2	-
3	Z^3 = T1 · T2	Z^4	Z^2	Z
4	Z^8 = T1 · T1	Z^3	Z^4	Z
5	Z^7 = T1 · T2	Z^8	Z^4	Z^3
6	Z^16 = T1 · T1	Z^7	Z^8	Z^3
7	Z^15 = T1 · T2	Z^15	Z^8	Z^7
.	.	.	.	.
.	.	.	.	.
499	Z^(2^250 - 1) = T1 · T2	Z^(2^250)		
500	-	Z^(2^250 - 1)	Z^(2^250)	Z^(2^249 - 1)
501	-	-	Z^(2^250)	Z^(2^250 - 1)

Table 11. Forward Multiplication Operation Schedule

Note that at cycle 4, 5, we will store Z^3 and Z^8 into X2 and Y2 registers for later usage.

For SELF state, the schedule of multiplier instance is below:

Cycle	Multiplier	Modp	T2 reg
0	Z^(2^251 - 2) = T2 · T2	-	-
1	-	Z^(2^251 - 2)	-
2	Z^(2^252 - 4) = T2 · T2	-	Z^(2^251 - 2)
3	-	Z^(2^252 - 4)	-
4	Z^(2^253 - 8) = T2 · T2	-	Z^(2^252 - 4)
5	-	Z^(2^253 - 8)	-
6	Z^(2^254 - 16) = T2 · T2	-	Z^(2^253 - 8)
7	-	Z^(2^254 - 16)	-
8	Z^(2^255 - 32) = T2 · T2	-	Z^(2^254 - 16)
9	-	Z^(2^255 - 32)	-
10	-	-	Z^(2^255 - 32)

Table 12. Self Multiplication Operation Schedule

For POST state, the schedule of multiplier instance is below:

Cycle	Multiplier	Modp	T1 reg
0	Z^11 = X2 · Y2	-	-
1	-	Z^11	-
2	Z^(2^255 - 21) = T1 · T2	-	Z^11
3	-	Z^(2^255 - 21)	-
4	-	-	Z^(2^255 - 21)

Table 13. Post Multiplication Operation Schedule

We have previously implemented Modular Inversion with algorithm in TA's code, which use a series of Num_Mult operation ins ALU. Therefore, we can also compare the speed of these two implementations:

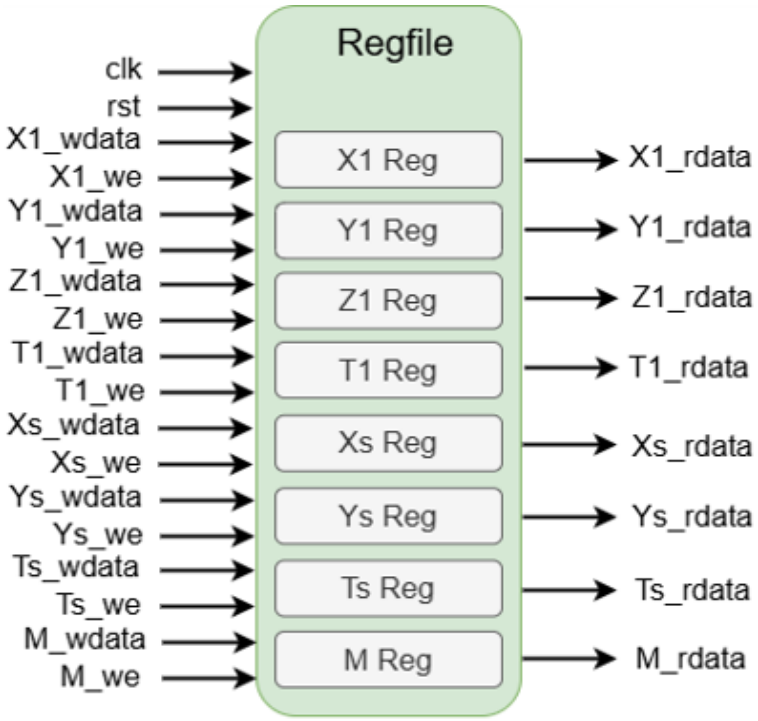
	TA's algorithm	Ours
Cycles (Modular Inversion)	1017	516
Cycles (Total)	4445	3944

Table 14. Hardware Speed Comparison of two Modular Inverse algorithms

We achieve approximately 2x speed on Modular Inverse compared with TA's algorithm, which justifies the efficiency of this algorithm from hardware perspective. Note that this number is based on pattern 0.

(3) Regfile

Regfile module contains eight 255-bit registers with write enable:



Xs, Ys, Ts, M stores the value related to input point and scalar. They won't be changed during the entire computation process and are only accessible by our System Control. Note that since Z (self.Z) of the input point is always equal to 1, we don't need to store it during the entire computation.

X1, Y1, Z1, T1 stores the intermediate value generated during the computation. Either System Control or ALU can access these four registers.

4. Performance Evaluation

In this part, we compare different version of design and perform optimization at different stages. We first discuss the speedup technique and the corresponded performance at RTL stage:

Version	# Cycle	Improvement	Description
v0	5921	-	Initial Version
v1	4699	26.0%	1. Add ALU arithmetic pre-fetch 2. Skip pre-calculation of LUTs in Point Add
v2	4445	33.2%	Add Mo = Z1 · Z1 pre-calculation for Point Double
v3	3944	50.1%	Use efficient algorithm of Modular Inversion

Table 15. Comparison of the number of cycles for different versions on Pattern 0

⇨ We use v3 as our final version in RTL Level Design.

For our final version of RTL, we also perform synthesis at different constraints. We then compare the performance of our different version of netlist:

Version	Clock Period (ns)	Area (mm^2)	Area* Time (mm^2 · ns)	Slack
v0	10	1.62342051	16.2342051	Met
v1	9.5	1.63068028	15.4914627	Met
v2	9	1.74052416	15.6646931	Met
v3	8.5	x	x	Un-met

Table 16. Netlist of different Synthesis Constraints

⇨ We use v1 as our final version in Gate Level Netlist

For our final version of Gate Level Netlist, we also perform P&R at different core utilization constraint:

Version	Core Utilization	Slack
v0	0.85	Met
v1	0.875	Met
v2	0.9	Un-met

Table 17. APR result at different Core Utilization constraint

⇨ We use v1 our final design after P&R

Finally, we use some metrics to evaluate our final chip design:

Metric	**Value**
# Cycle	**3944**
Clock period (ns)	**9.5**
Simulation Time (ns)	**37468**
Synthesis Area (mm^2)	**1.63068028**
Layout Area (mm^2)	**1.90926042**
Area * Time (mm^2· ns)	**71536.1694**

Table 18. Final Performance of our design

Note that the number is based on pattern 0.

5. Reference

[1] Final Project Note v3

[2] XUE Yiming, LIU Shurong, GUO Shuheng, LI Yan, and HU Cai'e, "High-speed hardware architecture design and implementation of Ed25519 signature verification algorithm," Journal on Communications, 2022.

[3] Bin YU, Hai HUANG , Zhiwei LIU, Shilei ZHAO, and Ning NA, "High-performance Hardware Architecture Design and Implementation of Ed25519 Algorithm," Journal of Electronics & Information Technology, 2021.

[4] Karatsuba algorithm (https://en.wikipedia.org/wiki/Karatsuba_algorithm)