4-bit Elliptic Curve Cryptography with Lopez-Dahab Projective Coordinates

Kirra Kotsenburg u1260664, Kadon Stimpson u1135047

May 1, 2024

1 Abstract

The goal of this project is to implement Elliptic Curve Cryptography using Lopez-Dahab formulas [1] for the projective coordinates used in point doubling and point addition. We will be utilizing Singular in several areas of this project. The first will be for finding all the elements in our Finite Field F_{2^4}, followed by substituting into and factorizing our elliptic curve formula to get the points of the curve. After we have done this we will need to find a generator point to generate all points on the curve (2P, 3P,...ect.). Once we have completed this step, we will utilize Quartus and ModelSim in order to create a hardware version of Point Doubling and Point Addition, the modules we make will be tested and compared to our outputs generated in Singular in order to verify correctness. The final step of our project will be to implement El Gamal encipherment in Singular and prove that the Lopez-Dahab formulas work for the function of encryption and decryption.

2 Introduction

Elliptic Curve Cryptography is a powerful tool in providing security and privacy. The aim of our project is to implement ECC while using a different coordinate system than studied in class. The formulas provided by Lopez-Dahab in [1] promises faster implementation of elliptic curve arithmetic over GF(2^4) which is greatly desired for hardware implementation where time is a concern.

The initial focus of this project will be in laying the ground work for our ECC implementation. That requires that we select an irreducible polynomial based on the amount of bits we want (4 in our case). Then we will find all our Finite Field's (F_{2^4}) elements in order to proceed to generating all the points on the curve. All of this is to be done in Singular.

The second focus of our project is to, after we have done our ground laying in Singular, write verilog modules for point doubling and point addition which requires additional modules for multiplication and addition, as well as testbenches to prove functionality.

Finally, to truly prove how well our ECC with Lopez-Dahab project works, we will add into a Singular file El Gamal and perform encryption and decryption. This will prove the success and usefulness of our project.

3 Implementation

3.1 Field Elements

Our selected irreducible polynomial: poly P = $\alpha^4 + \alpha + 1$

Elements of F_{16}	
Exponent	Polynomial
0	0
1	1
α^1	α
α^2	α^2
α^3	α^3
α^4	$\alpha + 1$
α^5	$\alpha^2 + \alpha$
α^6	$\alpha^3 + \alpha^2$
α^7	$\alpha^3 + \alpha + 1$
α^8	$\alpha^2 + 1$
α^9	$\alpha^3 + \alpha$
α^{10}	$\alpha^2 + \alpha + 1$
α^{11}	$\alpha^3 + \alpha^2 + \alpha$
α^{12}	$\alpha^3 + \alpha^2 + \alpha + 1$
α^{13}	$\alpha^3 + \alpha^2 + 1$
α^{14}	$\alpha^3 + 1$

3.2 Points Over the Elliptic Curve

For this section we will substitute and factorize the elliptic curve formula in order to get each element's matching factors which are points of the curve.

points		
0	(0, 1)	
α	No Factors	
α^2	No Factors	
α^3	$(\alpha^3, \alpha + 1)$	$(\alpha^3, \alpha^3 + \alpha + 1)$
α^4	No Factors	
α^5	$(\alpha^5, \alpha + 1)$	$(\alpha^5, \alpha^2 + 1)$
α^6	$(\alpha^6, \alpha^3 + \alpha^2)$	$(\alpha^6, 0)$
α^7	No Factors	
α^8	No Factors	
α^9	$(\alpha^9, \alpha + 1)$	$(\alpha^9, \alpha^3 + 1)$
α^{10}	$(\alpha^{10}, \alpha^3 + \alpha)$	$(\alpha^{10}, \alpha^3 + \alpha^2 + 1)$
α^{11}	No Factors	
α^{12}	$(\alpha^{12}, \alpha^2 + \alpha)$	$(\alpha^{12}, \alpha^3 + 1)$
α^{13}	No Factors	
α^{14}	No Factors	
1	$(1, \alpha^3)$	$(1, \alpha^3 + 1)$

Table 1: Points on the Curve

3.3 Lopez-Dahab Projective Coordinates

In our homework we used projective coordinates in hardware implementation in order to get rid of the division (which is multiplicative inverse) required to find the slope needed in the point doubling and point addition formulas. The conversion of our affine point was $x = X/Z$ and $y = Y/Z$ substituted into our elliptic curve formula.
This project utilizes the Lopez-Dahab formulas provided in [1] for faster implementation of elliptic curve arithmetic over GF(2^4). The substitution for x and y in the elliptic curve formula becomes $x = X/Z$ and $y = Y/Z^2$ which gives us Figure 1.

$$Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4 \ .$$

Figure 1: Projective Equation

The following two figures, Figure 2 and 3, are the formulas for point addition and doubling given by [1] and implemented in our design.

$$
\begin{aligned}
Z_2 &= Z_1^2 \cdot X_1^2 \ , \\
X_2 &= X_1^4 + b \cdot Z_1^4 \ , \\
Y_2 &= bZ_1^4 \cdot Z_2 + X_2 \cdot (aZ_2 + Y_1^2 + bZ_1^4) \ .
\end{aligned}
$$

Figure 2: LD Point Doubling Formula

$$
\begin{array}{lll}
A_0 = Y_1 \cdot Z_0^2 \ , & D = B_0 + B_1 \ , & H = C \cdot F \ , \\
A_1 = Y_0 \cdot Z_1^2 \ , & E = Z_0 \cdot Z_1 \ , & X_2 = C^2 + H + G \ , \\
B_0 = X_1 \cdot Z_0 \ , & F = D \cdot E \ , & I = D^2 \cdot B_0 \cdot E + X_2 \ , \\
B_1 = X_0 \cdot Z_1 \ , & Z_2 = F^2 \ , & J = D^2 \cdot A_0 + X_2 \ , \\
C = A_0 + A_1 \ , & G = D^2 \cdot (F + aE^2) \ , & Y_2 = H \cdot I + Z_2 \cdot J \ .
\end{array}
$$

Figure 3: LD Point Addition Formula

The Lopez-Dahab formula for the inverse of point P(X, Y, Z) is given by Figure 4.

$$
-P(X, Y, Z) = (X, XZ + Y, Z)
$$

Figure 4: Inverse of P

3.4 Enumerate All Points

Using Singular once more, we found a generative point for which we could generate all our points. The point doubling and addition in this section is based on the previous section's formulas taken from [1]. From this Singular file we generated the following points shown in Table 2. The table includes the projective points found and the normalized Affine point converted from the projective point.

Generative Point: $x1 = \alpha^3, y1 = \alpha^4, z1 = 1$

Generated Points		
P	Projective	Affine
P	$(\alpha^3, \alpha+1, 1)$	$(\alpha^3, \alpha+1)$
2P	$(\alpha^3+\alpha^2+\alpha, \alpha, \alpha^3+\alpha^2)$	$(\alpha^2+\alpha, \alpha+1)$
3P	$(\alpha^2+\alpha+1, \alpha, \alpha)$	$(\alpha^3+\alpha, \alpha^3+1)$
4P	$(\alpha^3+\alpha^2, \alpha^3+\alpha^2+\alpha, \alpha^3+\alpha^2)$	$(1, \alpha^3+1)$
5P	$(\alpha^3+\alpha^2+1, 0, \alpha^3+\alpha+1)$	$(\alpha^3+\alpha^2, 0)$
6P	$(\alpha^3+\alpha^2+\alpha+1, \alpha^2, \alpha^2)$	$(\alpha^2+\alpha+1, \alpha^3+\alpha^2+1)$
7P	$(\alpha^3+1, \alpha^3+\alpha, \alpha^2)$	$(\alpha^3+\alpha^2+\alpha+1, \alpha^2+\alpha)$
8P	$(0, \alpha^3+\alpha^2+\alpha, \alpha^3+\alpha^2+1)$	$(0, 1)$
9P	$(\alpha^2+\alpha+1, \alpha^2+\alpha+1, \alpha^3+\alpha^2+1)$	$(\alpha^3+\alpha^2+\alpha+1, \alpha^3+1)$
10P	$(\alpha^3+\alpha+1, \alpha^3, \alpha^3+\alpha^2+\alpha+1)$	$(\alpha^2+\alpha+1, \alpha^3+\alpha)$
11P	$(\alpha^3, 1, \alpha^3+\alpha^2+\alpha+1)$	$(\alpha^3+\alpha^2, \alpha^3+\alpha^2)$
12P	$(\alpha^3+\alpha+1, \alpha^2, \alpha^3+\alpha+1)$	$(1, \alpha^3)$
13P	$(\alpha^2+\alpha, \alpha^3+\alpha^2+\alpha, \alpha^3+\alpha^2+\alpha)$	$(\alpha^3+\alpha, \alpha+1)$
14P	$(\alpha^3+\alpha^2, \alpha^2+\alpha+1, \alpha)$	$(\alpha^2+\alpha, \alpha^2+1)$
15P	$(\alpha^3+1, \alpha^3+1, \alpha^3+\alpha^2+\alpha)$	$(\alpha^3, \alpha^3+\alpha+1)$
16P	$(m, 0, 0)$	O (infinity)

Table 2: Generated Points

3.5 Hardware Implementation of Point Doubling/Addition

In order to implement point doubling and addition in hardware, we first had to make the base modules for addition, multiplication, and squaring. Our multiplication and squaring modules are Mastrovito Multipliers with a further reduction of $(\mathrm{mod}(\alpha^4+\alpha+1))$. The Addition module performs simple bitwise XOR on two inputs.

With these three base modules we were able to design and implement our pointDoubling and pointAdd modules and replicate the Lopez-Dahab formulas for both.

Our testbench demonstrates these two modules working. Figure 5 shows the wave table of the testbench, the actual output R matching the expected_R for both Test 1 (point addition) and Test 2 (point doubling). The testbench also shows operations with the point at infinity. We can see that the modules correctly calculate the values for this point, as seen in tests 3 and 4.

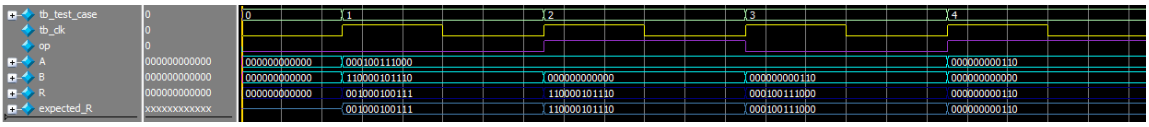

Figure 5: Testbench Results for Doubling and Addition

The RTL view of pointDouble in Figure 6 shows that we use 10 multipliers and 4 adders. In homework 3, when implementing the formulas provided in the

class slides our pointDouble module had 12 multipliers and 4 adders. Using Lopez-Dahab formulas we were able to remove two multipliers and retain the same functionality.

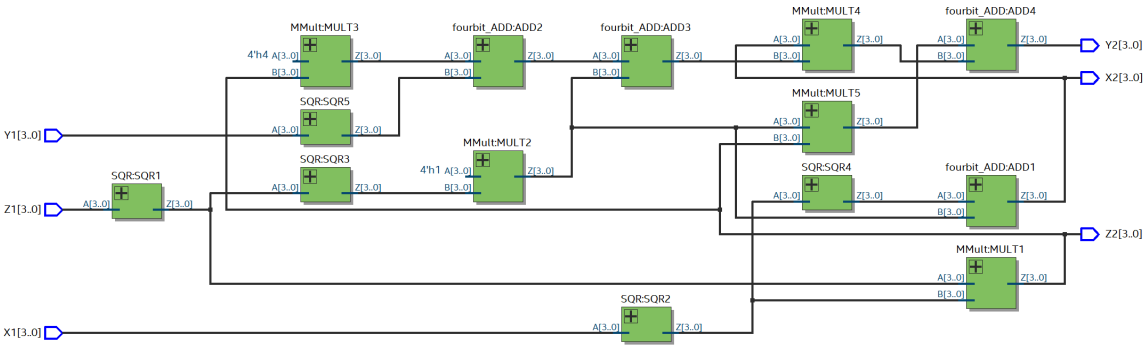

Figure 6: pointDouble RTL

The RTL view of pointADD in Figure 7 shows that there are 21 multipliers and 8 adders using the Lopez-Dahab formulas. We don't have an implementation of the point addition provided in the class slides, however, from running through the formula provided there we can say it should be around 14 multipliers and 7 adders, which is less than the pointADD with Lopez-Dahab formulas.

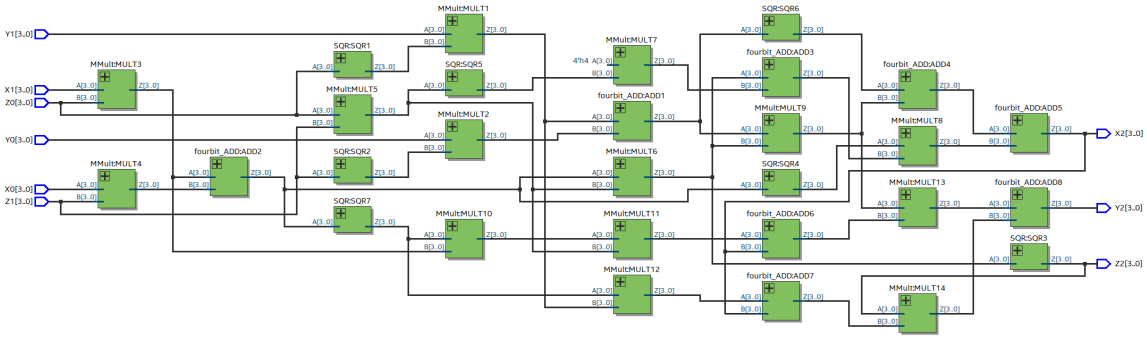

Figure 7: pointAdd RTL

3.6 El Gamal Encryption and Decryption in Singular

The final step of our project is to showcase our implementation of our point doubling and addition being used to successfully perform the encryption and decryption steps of El Gamal Encipherment. To do this we wrote another Singular File.
Figure 8 shows our public and private keys plus the plaintext that we want encrypted then decrypted. C1 and C2 are calculated and then decrypted. The result is a projective coordinate that we then normalize to be an affine point, which matches the input plaintext, showcasing the successful functionality of our El Gamal implementation with Lopez-Dahab coordinate formulas.

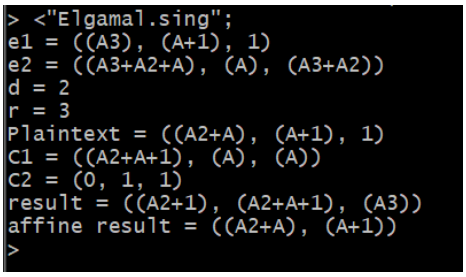

Figure 8: Singular Implementation of El Gamal Encipherment

Figure 9 provides an example of changing our private keys and the plaintext to show that the encipherment will work on other cases.

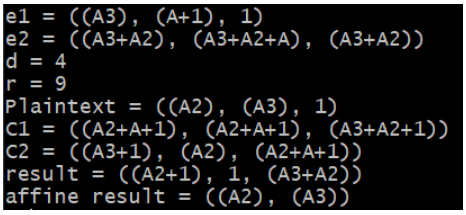

Figure 9: Test 2 of El Gamal Encipherment

4 Concepts Learned

- ECC - we solidified what we learned in class about this concept. We were able to test ourselves with more bits and understanding how different curves might have very different looking sets of points i.e. there were a lot more elements with no factors/points with our curve. This also helped with understanding the point at infinity.

- Lopez-Dahab - Understanding that different point systems could be used and might allow for better optimization over the standard coordinate system.

5 Contribution

The division of labor:

- Singular code:
  Kadon wrote the Singular files used in this project

- Verilog
  Kirra wrote the base modules for the multiplier and adder. She also wrote the pointDouble and pointADD modules. Kadon wrote testbenches for point doubling and addition and tweaked some of the code. He also created an ALU module for selection of point doubling or addition. Both team members performed debugging for this code, with Kadon being able to successfully have it running on his computer.

6 Conclusion

In conclusion, we were able to implement elliptic curve cryptography with the Lopez-Dahab formulas. Switching to the new formulas and using 4-bits allowed us to expand upon material taught in class in order to complete the project successfully. Our testbenches for the verilog code demonstrated correct functionality and we were able to move onto the El Gamal implementation in Singular. The code in that file demonstrated that we could encrypt plaintext with our newly formulated doubling and addition and then correctly decrypt back to the original plaintext. We were also able to use different values for keys and achieve the same results, showcasing the success of the project overall.

7 Resources

[1] López, J., Dahab, R. (1999). Improved Algorithms for Elliptic Curve Arithmetic in GF(2^n). In: Tavares, S., Meijer, H. (eds) Selected Areas in Cryptography. SAC 1998. Lecture Notes in Computer Science, vol 1556. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-48892-8_16