ABSTRACT

This project focuses on the development of a secure smart metering system using DLMS (Device Language Message Specification) Standards. The system employs advanced cryptographic algorithms, including AES-GCM (Advanced Encryption Standard - Galois/Counter Mode), SHA (Secure Hash Algorithm), and ECDH (Elliptic Curve Diffie-Hellman) for data encryption and key exchange. The implementation of the encryption process is done using Verilog code, which is uploaded into an FPGA (Field Programmable Gate Array) for hardware acceleration. The proposed system ensures secure and reliable communication between the smart meter and the central utility system while protecting against various cyber threats. The project demonstrates the practical application of advanced cryptographic techniques to improve the security and privacy of smart metering systems.

Keywords : Secure smart metering system, DLMS Standards, FPGA

ACKNOWLEDGEMENTS

We would like to express our deepest gratitude to the following people for guiding us through this course and without whom this project and the results achieved from it would not have reached completion.

Dr. Moorthi Sridharan, Assistant Professor, Department of Electronics and Electrical Engineering, for helping us and guiding us in the course of this project. Without his guidance, we would not have been able to successfully complete this project. His patience and genial attitude is and always will be a source of inspiration to us.

Dr Raja P , Associate Professor, Department of Electronics and Electrical Engineering, for his continuous support through constructive comments during project reviews.

Dr. Selvan M P, the Head of the Department, Department of Electronics and Electrical Engineering, for allowing us to avail the facilities at the department.

We are also thankful to the faculty and staff members of the Department of Electronics and Electrical Engineering, our individual parents and our friends for their constant support and help.

TABLE OF CONTENTS

Title	Page No.
ABSTRACT	i
ACKNOWLEDGEMENTS	ii
TABLE OF CONTENTS	iii
LIST OF FIGURES	vi
NOMENCLATURE	viii
CHAPTER 1 INTRODUCTION	1
1.1 Overview	1
1.2 Objectives	1
1.3 Background	1
1.4 Organization of Thesis	2
CHAPTER 2 Literature Review	4
CHAPTER 3 Proposed Methodology	6
3.1 Overall Block Diagram	6
3.2 AES	8
3.3 AES-GCM	12
3.3.1 Authenticated Encryption Function	12
3.4 SHA256	17
3.4.1 Message Padding	18
3.4.2 Message Schedule Computation	19
3.4.3 Hash State Initialization	19
3.4.4 Compression Function	20
3.4.5 Hash Output	21

3.5 ECDSA 22
3.5.1 Elliptic Curves 22
3.5.2 Group Laws for Elliptic Curves 22
3.5.3 Elliptic Curves in a Finite Field 25
3.5.4 Design Architecture 27
CHAPTER 4 Results and Discussions 31
4.1 Simulation 31
4.1.1 AES-GCM 31
4.1.2 SHA & ECDSA 33
4.2 Synthesis 34
4.2.1 AES-GCM 35
4.2.2 SHA 36
4.2.3 ECDSA 37
4.3 Implementation 38
4.3.1 AES-GCM 39
4.3.2 SHA 39
CHAPTER 5 Conclusion 42
5.1 Summary 42
5.2 Future Scope 42
REFERENCES 44
APPENDIX A CODE ATTACHMENTS 45
A.1 AES 45
A.1.1 TOP-MODULE 45
A.1.2 SUB-BYTES 47
A.1.3 MIX-COLUMNS 48
A.1.4 SHIFT-ROWS 49
A.1.5 FORWARD-SUBSTITURION 50
A.1.6 KEY GENERATION 55
A.1.7 ROUND-ITERATION 55

A.1.8 LAST ROUND 56
A.2 GCM 56
A.2.1 TOP-MODULE 56
A.2.2 ICB initialization 58
A.2.3 GCTR 58
A.2.4 GHASH 59
A.2.5 Multiplication in Galois Field 59
A.3 ECDSA 60
A.4 SHA256 88
A.4.1 SHA Top 88
A.4.2 SHA256 89
A.4.3 SHA Constants 95
A.4.4 SHA256 W Mem 96
A.5 UART 100
A.5.1 UART Top 100
A.5.2 Hex to ASCII 100
A.5.3 Debounce Signal 101
A.5.4 Transmitter 102

LIST OF FIGURES

3.1 Flow Diagram 6

3.2 ECDH 7

3.3 Flowchart for AES encryption 9

3.4 Flowchart for Key Expansion 9

3.5 SubBytes 10

3.6 ShiftRows 10

3.7 MixColumns 10

3.8 AddRoundKeys 11

3.9 GCTR Algorithm 14

3.10 GHASH Algorithm 15

3.11 AES-GCM Algorithm 16

3.12 SHA256 Overview 18

3.13 SHA Message Padding 19

3.14 Elliptic Curves 22

3.15 Geometric Addition 23

3.16 Elliptic Curves in Finite Fields 26

4.1 AES Simulation Result 31

4.2 AES Verification 32

4.3 AES-GCM Simulation 32

4.4 SHA ECDSA Simulations 34

4.5 SHA ECDSA Verification 34

4.6 AES-GCM RTL Schematic 35

4.7 AES-GCM Utilization 35

4.8 AES-GCM Power 36
4.9 SHA256 Utilization Report 36
4.10 SHA256 Schematics 37
4.11 ECDSA Schematics 37
4.12 AES-GCM FPGA 39
4.13 SHA256 FPGA implementation 39
4.14 SHA256 FPGA Enable signal 40
4.15 SHA256 UART Output 40

NOMENCLATURE

AES	Advanced Encryption Standard
ASCII	American Standard Code for Information Interchange
DLMS	Device Language Message Specification
ECDH	Elliptic Curve Diffie Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
FPGA	Field Programmabe Gate Arrays
GCM	Galois Counter Mode
MD5	Message Digest Algorithm
NIST	National Institute of Standards and Technology
NSA	National Security Agency
RSA	Rivest Shamir Adleman
SCADA	Supervisory Control And Data Acquisition
SHA	Secure Hash Algorithm
TLS	Transport Layer Security
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus

CHAPTER 1

INTRODUCTION

1.1 Overview

The project aims to enhance the security of data transmission in smart meters that use DLMS standards by implementing encryption algorithms such as AES-GCM, SHA, and ECDSA using Verilog, a hardware description language used in digital circuit design.

The primary objective is to enable secure communication between smart meters and the utility company, preventing unauthorized access to sensitive information such as energy usage and billing data. The encryption algorithms will be integrated into an FPGA device, providing hardware-based encryption for smart meters.

The project represents an innovative approach to improving the security of data transmission in smart meters, with the potential to enhance the reliability of energy usage data and protect consumer privacy, which are crucial aspects of smart grid technology. The implementation of the encryption algorithms in Verilog and integration into the FPGA device ensures that the data transmitted between smart meters and the utility company remains secure and private.

1.2 Objectives

The primary objective of this project has been to ensure the security of data communications in smart meters. In order to secure the transmission in smart meters, the project has studied different communication protocol standards and gained an understanding of how data communication occurs, including the encryption standards used. The final objective of the project has been to implement the solution in the FPGA and conduct tests to assess the results. By implementing the solution in the FPGA, the project has validated the design's feasibility for implementation in hardware.

1.3 Background

Smart Meters are deployed around the country due to its various benefits such as reduction in cost of overproducing or under producing, Detailed Electricity consumption report allowing consumers to make informed decisions, Detection of faulty appliances, Energy theft detection, and enhancements in grid reliability. All these

data which is provided by the smart meters has to be transmitted to the utility either via power line communication, wireless communication or wired communication. During the transmission of data there is a possibility of data theft, and data tampering which is catastrophic and makes the smart meter system unreliable. In order to ensure secure transmission of data through smart meters we need to encrypt the data before its transmitted.

The communication protocol which we have considered for our project is Device Language Message Specification (DLMS). This is a globally recognized protocol for communication that defines the semantics and the syntax of a language for data exchange with smart devices in an interoperable, efficient and secure manner. It supports applications such as remote meter reading, remote control, and value-added services for metering any kind of energy, such as electricity, water, gas or heat. To specify data exchange with smart devices DLMS uses a three-step approach:

1. Modelling i.e. the semantics, defined with COSEM objects.
2. Messaging i.e. the syntax, defined with DLMS services.
3. Transport, defined with communication profiles.

The DLMS security mechanisms include Entity authentication, Role-based access, Message protection, Data protection, Firmware upgrade, Communication port protection, Security and efficiency, Security logs, Security suites, Security policy, and Security Keys. The security algorithms used in DLMS provide confidentiality, Data integrity, Digital Signature, Key transport, Key agreement, and Hash algorithms. There are three security suites provided by DLMS. We will be implementing the level 1 security suite of DLMS in our project. In this suite AES-GCM 128 is used for Authenticated encryption, ECDSA with P-256 is used for Digital signature, ECDH with P-256 for Key agreement, SHA-256 for Hash generation, AES-128 Key-Wrap for Key transport and V.44 for Compression.

1.4 Organization of Thesis

The thesis begins with an introduction that outlines the purpose and objectives of the research, followed by a section that briefly discusses the relevant background literature on smart meter technology and encryption techniques. The literature review chapter provides a critical analysis of the existing literature in the field of study, while the methodology chapter describes the methods used to design the encryption module and the organization of each module.

The thesis also includes a chapter on synthesis and implementation, where the designed encryption module is tested through simulation and implementation. The

synthesis report provides a detailed description of the synthesis process, including the design specifications and the parameters used in the synthesis tool. The implementation report presents the results of the simulation and implementation, using relevant visual aids such as snippets and reports to demonstrate the feasibility of the design.

The results chapter presents the findings of the simulation and implementation, highlighting the success of the encryption module design. The conclusion provides a summary of the main findings and their implications, highlighting the contributions of the research to the field of study. Finally, a comprehensive list of the references cited throughout the research is included at the end of the thesis.

CHAPTER 2

Literature Review

Smart meters have been devices that have measured energy consumption in real-time and have provided accurate and up-to-date information about energy usage. They have offered several benefits, including more accurate billing, improved energy management, and improved grid efficiency [1]. By providing real-time data on energy usage patterns, smart meters have helped customers reduce their energy consumption. Smart meters have been essential for modernizing the energy system and have provided customers with more accurate and useful information about their energy consumption.

In order to have achieved these functionalities, the smart meter has had to have bidirectional communication between the household and the service provider. The data that has been transmitted has contained sensitive information about a household's energy consumption, including how much energy has been used and when it has been used. This data has been able to reveal details about a household's behavior and routine, which could be used to infer personal information about the occupants. Encrypting smart meter data has helped to prevent unauthorized access and tampering, as well as protected against interception and eavesdropping during transmission [2]. It has also ensured that the data has remained confidential, which has been important for protecting the privacy of households and complying with data protection regulations.

Encryption of data has been able to be done in two ways: symmetric and asymmetric manner. The basic difference between the two encryption methods has been that in symmetric cipher, the same key has been used to encrypt and decrypt the message, while in asymmetric cipher, the public key has been used to encrypt the data, and the private key has been used to decrypt the data. For symmetric cipher, both the parties, the sender and receiver, have had to know the key, and it has had to be kept as a secret, whereas in asymmetric cipher, the public key has been shared among everyone, and the private key has been kept as a secret.

Either symmetric or asymmetric cipher has not been able to be used alone because of the drawbacks it has possessed. Symmetric cipher has been faster, more efficient, and secure, but the key distribution process has added more complexity and may have compromised the security. On the other hand, asymmetric cipher has not had the problem of key distribution, but it has been slower and inefficient. So, from the paper [3], we have come to the conclusion that hybrid mode of ciphers

has been able to be used to get the best of both worlds.

Here, symmetric algorithm Blowfish has been used to encrypt the main smart meter data, which has been faster and efficient, and the key for the symmetric cipher has been encrypted with the asymmetric algorithm RSA, eliminating the key distribution problems. Along with these encryption algorithms, MD5 and RSA digital signature have been used to provide more authenticity to the data. This hybrid cryptographic stack has formed the base for further studies. Even though multiple algorithms have been used, this stack has been a preliminary option without much optimization for a specific purpose.

Cryptography for SCADA [4] has provided a more optimum, efficient, and specific solution for data encryption for SCADA systems. The proposed methodology has also used a hybrid cryptographic stack with some changes to make it work for specific SCADA applications. This stack has used AES symmetric cipher instead of Blowfish because it has been more secure and recognized as a standard. And for asymmetric cipher, ECC has been preferred over RSA because for smaller size research has proved that ECC has been more efficient than RSA. The hashing algorithm used has remained the same, MD5.

DLMS has provided three different encryption standards that can be used for smart meter data encryption [5]. Out of which, level 1 security suite has been chosen for implementation because the standard mentioned in DLMS has been specific for smart meter, whereas in [4], it has been a general stack for SCADA system. And level 1 security suite has been efficient and has been a little complex to implement.

CHAPTER 3

Proposed Methodology

3.1 Overall Block Diagram

The encryption standard utilized for smart meter encryption is the DLMS Level 1 standard, which has multiple encryption algorithms to ensure secure transmission of contents. In this project, a flow diagram has been designed to depict the Level 1 suite process. The first step in this encryption process has been ECDH (Elliptic-curve Diffie–Hellman), which is a key-agreement protocol that defines how keys should be generated and exchanged between parties. This step has been essential as the following algorithms, namely AES and ECDSA, require key and other parameters to be known by both sender and receiver for proper functionality.

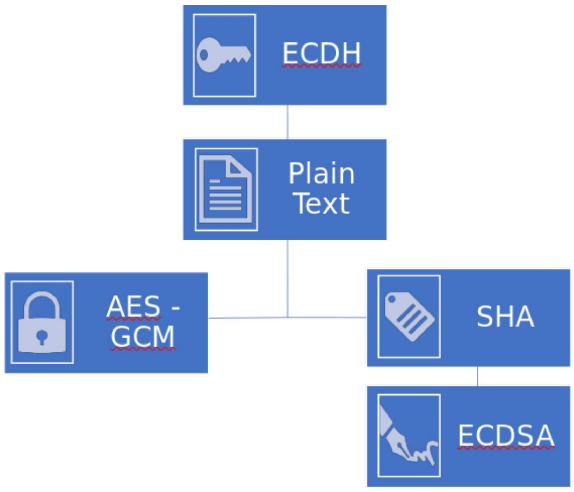


Figure 3.1: Flow Diagram

The AES (Advanced Encryption Standard) has been a widely used symmetric-key block cipher encryption algorithm to secure data by encrypting it with a key. This algorithm is considered to be highly secure due to its resistance to various types of attacks, including brute force attacks. The AES algorithm has used a symmetric key, which means that the same key has been used for both encryption and decryption. Hence, the sender and receiver have had to know the key for secure communication.

ECDSA (Elliptic Curve Digital Signature Algorithm) has been a cryptographic digital signature algorithm utilized to ensure the authenticity, integrity, and non-repudiation of data. It has been based on the mathematical properties of elliptic

curves and has been a widely used public-key algorithm. ECDSA has worked by using a private key to sign a message, which generates a digital signature. This signature has then been verified using the corresponding public key, which has been published and available to anyone. The signature has provided proof that the message was signed by someone with access to the private key and has not been tampered with since it was signed. ECDSA has been implementable only if curve parameters have been known to both sides. Thus, key transport should have occurred, and other parameters should have been known to both sides for the cryptographic stack to work efficiently.

ECDH (Elliptic-curve Diffie–Hellman) is a key-agreement protocol that enables two parties (typically Alice and Bob) to exchange information securely without the risk of interception by a third party (known as the Man In the Middle). This principle is often used in the TLS protocol to ensure secure communication between parties.

- To begin the ECDH process, both Alice and Bob generate their own private and public keys using the same domain parameters, such as the same base point on the same elliptic curve on the same finite field. Alice's private and public keys are denoted as and respectively, while Bob's private and public keys are denoted as and respectively.

- Next, Alice and Bob exchange their public keys and over an insecure channel. The Man In the Middle may intercept and , but they would not be able to obtain or without solving the discrete logarithm problem.

- Using her private key and Bob's public key, Alice calculates , while Bob calculates using his own private key and Alice's public key. Notably, the shared secret is the same for both Alice and Bob, and it cannot be discovered by the Man In the Middle, who only knows and (together with the other domain parameters). This problem is known as the Diffie-Hellman problem, and it is solved by ECDH.

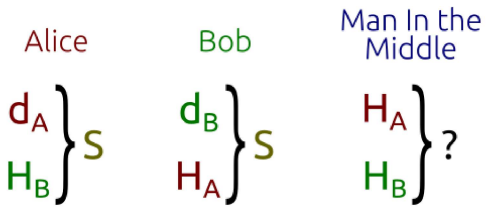


Figure 3.2: ECDH

The Diffie-Hellman problem for elliptic curves is considered to be a challenging problem in cryptography. While no mathematical proofs are available, it is believed

to be as difficult as the discrete logarithm problem. However, it cannot be more difficult than the logarithm problem since solving the logarithm problem provides a way to solve the Diffie-Hellman problem. In comparison to other key exchange protocols such as RSA or Diffie-Hellman (DH), the ECDH algorithm offers several advantages. It enables smaller key sizes, faster processing times, and stronger security for the same key size. As a result, it has become a popular choice for secure communication in various applications, including secure messaging and virtual private networks (VPNs).

In our current project, we assume that the ECDH key exchange algorithm has already been completed, and both parties have obtained the required keys and parameters. With the keys and other parameters known, we can focus on the encryption algorithms without any concerns about the key exchange process. This allows us to work more efficiently towards ensuring secure communication through the implementation of the DLMS Level 1 suite.

3.2 AES

AES is a widely used symmetric-key encryption algorithm. It was first introduced by the National Institute of Standards and Technology in 2001 and is now considered to be one of the most secure encryption algorithms available. AES is used to protect sensitive data, such as credit card numbers, passwords, and other personal information, in various applications, including online banking, e-commerce, and file encryption. The AES encryption algorithm operates on fixed block sizes of 128 bits and supports key sizes of 128, 192, and 256 bits. AES is widely adopted due to its high level of security, efficiency, and scalability.

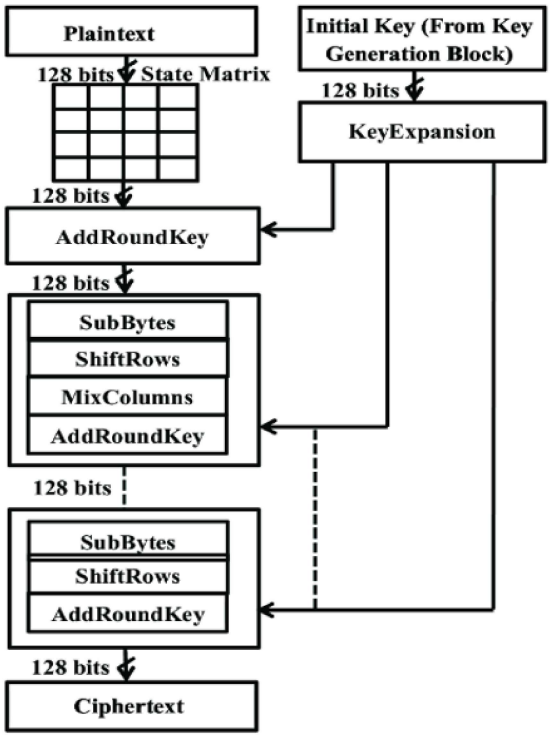


Figure 3.3: Flowchart for AES encryption

The AES algorithm works in a series of steps that includes key expansion, initial round, main rounds, and final round. Here's a brief explanation of each step:

1. Key expansion: In this step, the original key is expanded into a set of subkeys. These subkeys are generated through a key schedule that uses a combination of substitution and permutation operations.

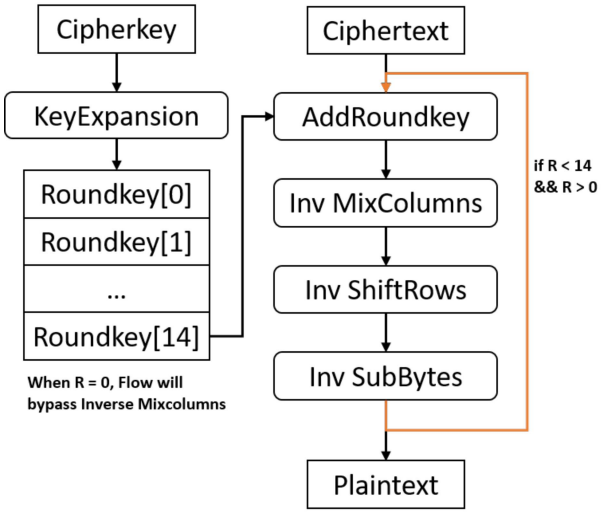


Figure 3.4: Flowchart for Key Expansion

2. Initial round: The initial round involves the transformation of the input data using a key generated from the key expansion step. This is done through a

process called ”AddRoundKey,” where each byte of the input data is combined with a corresponding byte of the subkey.

3. Main rounds: The main rounds involve a series of four different transformations: SubBytes, ShiftRows, MixColumns, and AddRoundKey.

   (a) In the SubBytes step, each byte of the input data is replaced with a corresponding byte from a fixed substitution table.

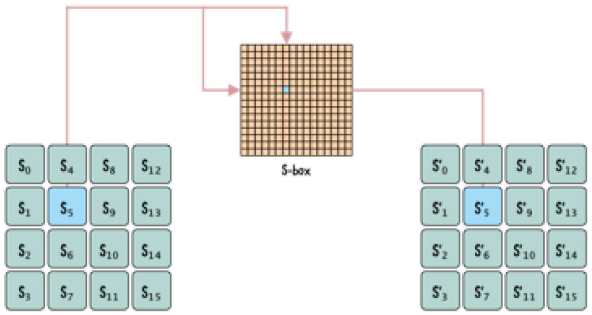


Figure 3.5: SubBytes

   (b) In the ShiftRows step, the rows of the input data are shifted by a certain number of bytes.

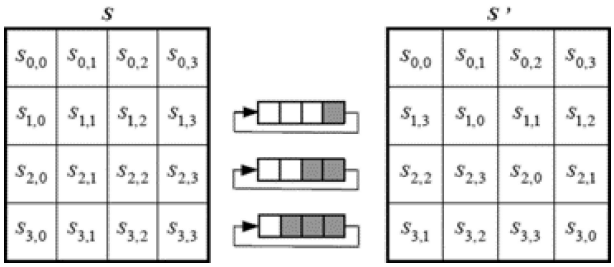


Figure 3.6: ShiftRows

   (c) In the MixColumns step, each column of the input data is multiplied with a fixed matrix.

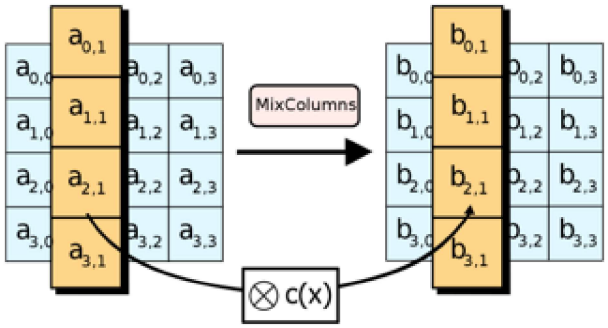


Figure 3.7: MixColumns

(d) Finally, in the AddRoundKey step, the input data is combined with a subkey.

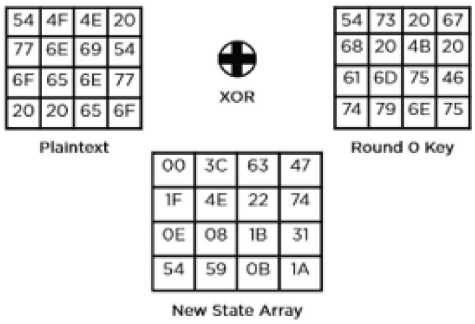


Figure 3.8: AddRoundKeys

4. Final round: The final round is similar to the initial round, but it omits the MixColumns step. The final round transforms the data using a key generated from the key expansion step and outputs the encrypted data.

In summary, the AES algorithm uses a combination of substitution, permutation, and multiplication operations to transform the input data using a key generated from the key expansion step. This process of encryption makes it very difficult for an attacker to reverse the encryption and obtain the original data without the proper key. Some of the uses for this encryption standard include

1. Secure Communication: AES is used to encrypt data transmitted over the internet, such as email, instant messaging, and online transactions. It is commonly used in protocols such as SSL/TLS, SSH, and IPSec to secure communications between servers and clients.

2. Data Storage: AES is used to encrypt data stored on hard drives, USB drives, and other storage devices to protect against unauthorized access in case of theft or loss. Many operating systems, including Windows and macOS, use AES encryption to secure user data.

3. Military and Government Applications: AES is used by the US government and military to protect classified information. It is also used by other governments and organizations to protect sensitive information.

4. Financial Transactions: AES is used to secure financial transactions, such as online banking and credit card transactions, to prevent unauthorized access and fraud.

5. Digital Rights Management: AES is used in digital rights management (DRM) systems to protect copyrighted material from unauthorized distribution and piracy.

Here is a pseudo-code for the encryption process:

3.3 AES-GCM

AES-GCM (Advanced Encryption Standard in Galois Counter Mode of Operation), GCM provides assurance of the confidentiality of data using a variation of the Counter mode of operation for encryption. GCM provides assurance of the authenticity of the confidential data (up to about 64 gigabytes per invocation) using a universal hash function that is defined over a binary Galois (i.e., finite) field. GCM can also provide authentication assurance for additional data (of practically unlimited length per invocation) that is not encrypted. The Elements of the GCM include Block Cipher (AES), Authenticated Encryption and Decryption function. In our project we will be implementing the Authenticated Encryption function for Smart Meters. The Authenticated decryption function can be used for verification. [6]

3.3.1 Authenticated Encryption Function

Initially the function requires the selection of a block cipher and key. The input data for authenticated encryption function are Plaintext (P), Additional Authenticated Data(AAD), and Initialization Vector (IV). The plaintext and the AAD are the two categories of data that GCM protects. GCM protects the authenticity of the plaintext and the AAD. It also protects the confidentiality of the plaintext, while AAD is left in the clear. The IV is essentially a nonce, a value that is unique within the specified context, which determines an invocation of the authenticated encryption function on the input data to be protected.
The output data from the function include Ciphertext (C), and an Authentication Tag (t). The bit length of the ciphertext is same as the plaintext. The bit length of the Authentication tag , denoted t, is a security parameter, varies from 128, 120, 112, 104, or 96. For some Application tag length can be 64 or 32.

There are some mathematical algorithms that are used as a part of this function which include the following :-

Incrementing Function

For a positive integer s and a bit string X such that $len(X)s$, let the s-bit incrementing function, denoted $inc_s(X)$, be defined as follows:

$$inc_s(X) = MSB_{len(X)-s}(X)||[int(LSB_s(X)) + 1 mod 2^s]_s$$

It is a 32 bit incrementing function, the function increments the right-most 32 bits and left-most $len(X) - 32$ bits remain unchanged. [7]

Multiplication Operation ($X \bullet Y$)

The $\bullet$ operation on (pairs of) the 2^{128} possible blocks corresponds to the multiplication operation for the binary Galois (finite) field of 2^{128} elements. The fixed block,R $(11100001||0^{120})$, determines a representation of this field as the modular multiplication of binary polynomials of degree less than 128.
Consider R to be a bit string $(11100001||0^{120})$. Given two input blocks X and Y, this algorithm computes a "product" block, denoted as X $\bullet$ Y.

Pseudo code

- Let $x_0x_1...x_{127}$ denote the sequence of bits in X.
- Let Z_0 = 0^{128} and V_0 = Y.
- For i = 0 to 127, calculate blocks Z_{i+1} and V_{i+1} as follows :

$$Z_{i+1} = \begin{cases} \mathrm{Z}_i & \text{if } x_i = 0 \\ \mathrm{Z}_i \oplus V_i & \text{if } x_i = 1. \end{cases}$$

$$V_{i+1} = \begin{cases} \mathrm{V}_i >> 1 & \text{if } LSB_1(\mathrm{V}_i) = 0 \\ (\mathrm{V}_i >> 1) \oplus R & \text{if } LSB_1(V_i) = 1 \end{cases}$$

- Return Z_{128}

This Encryption function uses two main functions GCTR and GHASH. The GCTR function is used for generating the Cipher Text and GHASH function is used for generating the authentication tag.

GCTR Function

The input required for this function is Initial Counter Block (ICB), Plaintext(X). The ICB is generated by concatenating the Initialization Vector (IV) with 31 zero

bits and a logic high bit. The output(Y) generated will have the same bit length as the input Plaintext(X).

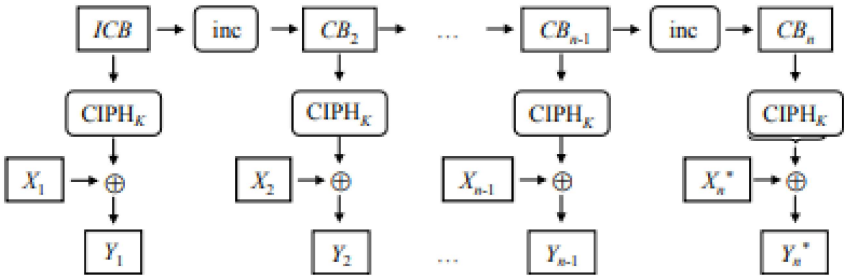


Figure 3.9: GCTR Algorithm

The input string of arbitrary length is partitioned into a sequence of blocks to the greatest extent possible, so that only the rightmost string in the sequence may be a "partial" block. The 32-bit incrementing function is iterated on the initial counter block input to generate a sequence of counter blocks; the input block is the first block of the sequence. The block cipher is applied to the counter blocks and the results are XORed with the corresponding blocks (or partial block) of the partition of the input string. The sequence of results is concatenated to form the output. The output will be the encrypted Ciphertext.

Pseudo Code

- If X is the empty string, then return the empty string as Y.
- Let $n = [len(X)/128]$.
- Let $X1, X2, ..., X_{n-1}, Xn^*$ denote the unique sequence of bit strings such that

$$X = X1||X2||....||X_{n-1}||Xn$$
$$X1, X2, ..., X_{n-1} \text{ are complete blocks}$$

- Let $CB_1 = ICB$.
- For $i = 2$ to n, Let $CB_i = inc_{32}(CB_{i-1})$.
- For $I = 1$ to $(n-1)$, let $Y_i = X_i \oplus CIPH_K(CB_i)$.
- Let $Y_n^* = X_n^* \oplus MSB_{len(Xn)}(CIPH_K(CB_n))$.

GHASH Function

The input required for this function is input string (Ciphertext from GCTR) and a hash subkey. The hash subkey is the output generated by passing 128 zero bits as input to AES algorithm. The output (T) generated from GHASH will be of 128 bits.

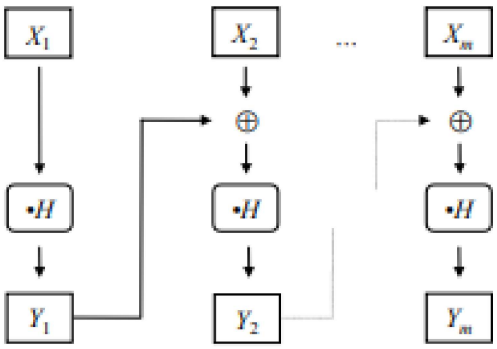


Figure 3.10: GHASH Algorithm

The GHASH function calculates $X1 \bullet Hm \oplus X2 \bullet Hm-1 \oplus ... \oplus X_{m-1} \bullet H2 \oplus Xm \bullet H$. Here the $\bullet$ function represents the multiplication algorithm. The GHASH function is illustrated in Figure, without the zero block, Y0, whose exclusive-OR with X1 does not change X1.

Pseudo Code

- Let $X_1, X_2, ..., X_{m-1}, X_m$ denote the unique sequence of blocks such that $X = X_1||X_2||...||X_{m-1}||X_m$.
- Let Y0 be the "zero block," 0^{128}.
- For $i = 1, ..., m$, let $Yi = (Yi-1 \oplus Xi) \bullet H$.
- Return Y_m.

Algorithm for Authenticated Encryption function

The hash subkey for the GHASH function is generated by applying the block cipher to the "zero" block. The pre-counter block (ICB) is generated from the IV. In particular, when the length of the IV is 96 bits, then the padding string 031——1 is appended to the IV to form the pre-counter block. In Step 3, the 32-bit incrementing function is applied to the pre-counter block to produce the initial counter block for an invocation of the GCTR function on the plaintext. The output of this invocation of the GCTR function is the ciphertext. The AAD and the ciphertext are each appended with the minimum number of '0' bits, possibly none, so that the bit

lengths of the resulting strings are multiples of the block size. The concatenation of these strings is appended with the 64-bit representations of the lengths of the AAD and the ciphertext, and the GHASH function is applied to the result to produce a single output block. This output block is encrypted using the GCTR function with pre-counter block that was generated earlier. The result is truncated to the specified tag length to form the authentication tag. The Ciphertext and tag are returned as the output.

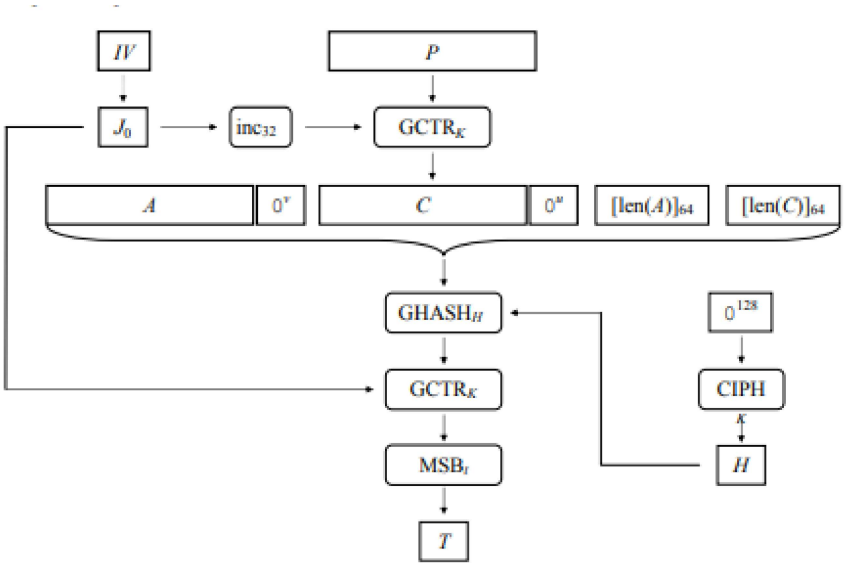


Figure 3.11: AES-GCM Algorithm

Pseudo Code

- Let $H = CIPH_K(0^{128})$.
- Define a block, ICB, as follows:
  If $len(IV) = 96$, then let $J_0 = IV||0^{31}||1$.
- Let $C = GCTR_K(inc_{32}(ICB), P)$.
- Let $u = 128.[len(C)/128]–len(C)$ and let v $= 128.[len(A)/128]–len(A)$.
- Define a block, S, as follows:

$$\text{S} = GHASH_H(A||0_v||C||0_u||[len(A)]_{64}||[len(C)]_{64}).$$

- Let $T = MSB_t(GCTR_K(ICB, S))$.
- Return (C, T).

3.4 SHA256

The SHA256 algorithm is a cryptographic hash function that generates a fixed-length hash value of 256 bits from a given input data. It belongs to the SHA-2 family of hash functions, which were developed by the United States National Security Agency (NSA) and published by the National Institute of Standards and Technology (NIST) in 2001. It is widely used and considered a secure hash function in various security applications, such as digital signatures and password storage. The algorithm operates by dividing the input data into blocks and then processing each block through a series of mathematical operations. The resulting 256-bit hash value is unique to the input data and is virtually impossible to recreate the input data for a particular hash value.

One of the essential characteristics of SHA256 is that it is a one-way function, which means that obtaining the original input data from the hash value is infeasible. Additionally, even a minor modification in the input data generates a distinct hash value, making it useful for detecting data tampering or corruption. The main steps followed in the SHA256 algorithm are as follows.

- Message Padding: The input message is padded to make its length a multiple of 512 bits.
- Message Parsing: The padded message is parsed into 512-bit blocks.
- Message Schedule Computation: For each block, a message schedule of 64 32-bit words is computed from the block's 512 bits.
- Hash State Initialization: The hash state, consisting of eight 32-bit words, is initialized to fixed constants.
- Compression Function: For each block, the compression function is applied to the current hash state and the message schedule.
- Hash Output: After all blocks have been processed, the final hash value is obtained by concatenating the eight 32-bit words of the final hash state.

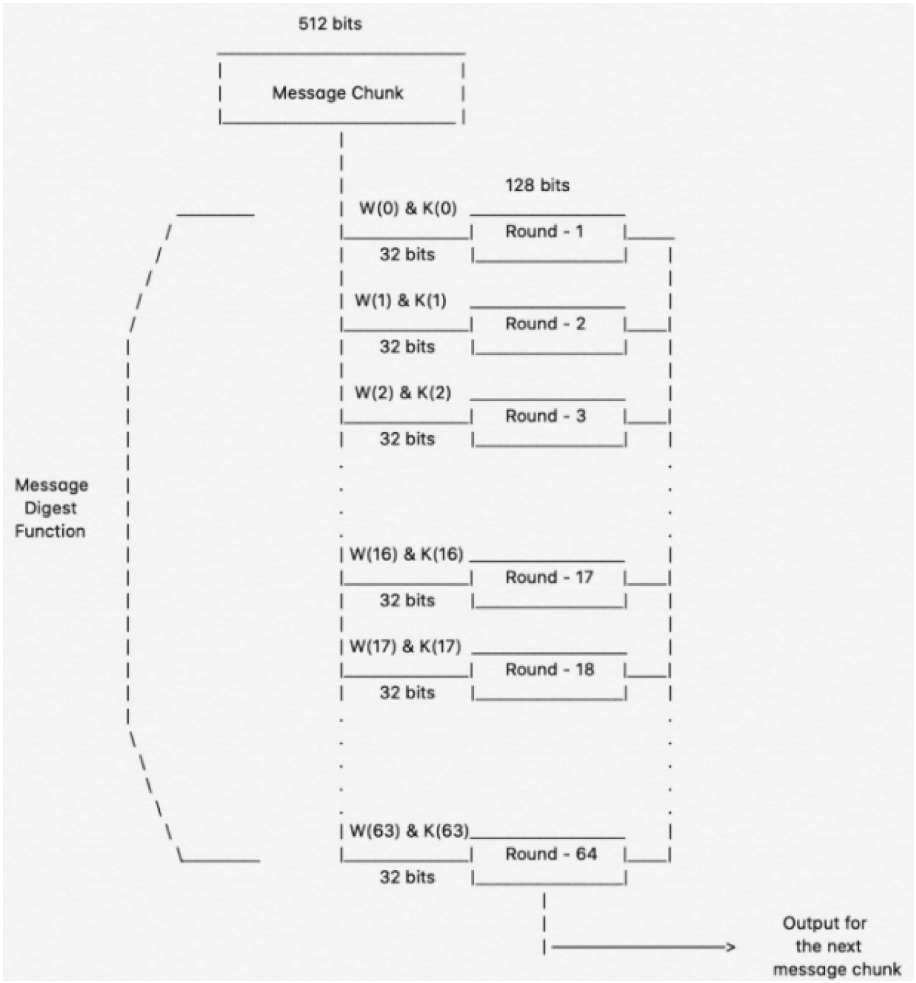


Figure 3.12: SHA256 Overview

3.4.1 Message Padding

Before beginning the hash computation, the message, M, has to be padded to ensure that the padded message is a multiple of 512 bits, which can be processed by SHA256. To achieve this, the bit ”1” is added to the end of the message, followed by the addition of k zero bits, where k is the smallest non-negative solution to the equation $l + 1 + k \equiv 448 \bmod 512$, where l is the length of the message in bits. Finally, the 64-bit block equal to the number l expressed using a binary representation is appended to the message. For instance, if the message is ”abc” with a length of 8x3=24 bits, it is padded with one bit followed by 423 zero bits, and then the message length is added to form a 512-bit padded message.

In this project, the padded message of the data that needed to be transmitted has been stored in the registers of FPGA, assuming that the smart meter has sent the data and padded it accordingly. After a message has been padded, it must be parsed into N m-bit blocks before the hash computation can begin. So the 512 bits of the input block may be expressed as sixteen 32-bit words

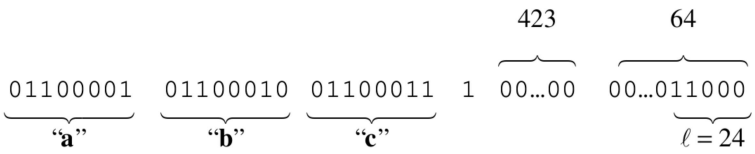


Figure 3.13: SHA Message Padding

3.4.2 Message Schedule Computation

After dividing the input message into 16 32-bit blocks, the message scheduling function is applied to create 64 32-bit blocks, which are then used for further data processing.

let us assume the parsed message as M0-M15

let us assume the message schedules as W0-W63

Message schedule function does the following process

- The first 16 message schedules (W0-W15) are copied directly from the initial 16 parsed message blocks (M0-M15).
- The last 48 message schedules (W16-W63) are computed using this function $W_n = \sigma_1(W_{n-2}) + W_{n-7} + \sigma_0(W_{n-15}) + W_{n-16}$

+ means Addition modulo 2^w

σ_0 means function that has some rotation and shift operation

σ_1 means function that has some rotation and shift operation

W_n means nth term of message schedule

In the hardware design, there is a separate module called SHA256_W_mem, which performs the message scheduling operation. This module takes the padded message as input and generates the message schedule by executing the aforementioned functions, and stores them in registers. When the message schedule number is provided as input, it retrieves the corresponding message schedule register value and provides it as output.

3.4.3 Hash State Initialization

The SHA256 algorithm employs eight initial hash values, which are referred to as the ”state,” to create the ultimate hash value by merging them with the input message. These hash values are derived from the initial 32 bits of the fractional parts of the square roots of the first eight prime numbers (2, 3, 5, 7, 11, 13, 17, 19) and are used to initialize the variables as the initial hash value. While processing the first 512 bits block at the start of the SHA process, the SHA256 module initializes

the Hash value registers, which comprise of eight 32-bit registers. However, for the subsequent block(s), the register value retains the result of the previous block.

H0 = 6a09e667

H1 = bb67ae85

H2 = 3c6ef372

H3 = a54ff53a

H4 = 510e527f

H5 = 9b05688c

H6 = 1f83d9ab

H7 = 5be0cd19

3.4.4 Compression Function

SHA256 employs 64 constant 32-bit words, each of which is stored in a module called SHA256_constants. When the module is provided with the input of a specific constant register number, it produces the corresponding constant register value as output. The module contains the sequence of constant values used in SHA256.

428a2f98 71374491 b5c0fbcf e9b5dba5 3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3 72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc 2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7 c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13 650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3 d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08 2748774c 34b0bcb5 391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3
748f82ee 78a5636f 84c87814 8cc70208 90befffa a4506ceb bef9a3f7 c67178f2

During the SHA256 hashing process, a total of 64 rounds are executed using the SHA256 constants and message schedules generated. Each round involves passing one constant value and one message schedule to a function that processes them according to the SHA256 algorithm.

For n = 0 to 63

{

T1 = h + $\sum_1^{256}$(e) + Ch(e, f , g) + K_n + W_n

T2 = $\sum_0^{256}$(a) + Maj(a, b , c)

h = g

g = f

f = e

e = d + T1

d = c
c = b
b = a
a = T1 + T2
}

+ means Addition modulo 2^w
$\sum_0^{256}$ means function that has some rotation operation
$\sum_1^{256}$ means function that has some rotation operation
K_n means nth term of SHA256 constants
W_n means nth term of message schedule
Ch(e, f, g) and Maj(a, b, c) are combinational functions

The eight working variables, namely a, b, c, d, e, f, g, and h, are initialized with the stored hash value of the particular block in the SHA256 module. These variables are updated in each round, and the final values of the working variables after 64 rounds are added to the previous or stored hash value to generate the hash value for that block. All of these operations are designed in the SHA256 module.

3.4.5 Hash Output

The SHA256 algorithm applies the described operations to each 512-bit block sequentially, where the output of each block is used as input for the subsequent block. After processing all the blocks, the eight hash values of the initial blocks are combined with the output of the final block to yield the final hash value. The SHA_digest_valid signal in the SHA_top module goes high to indicate the completion of the SHA256 process and the production of the final output for the final block.

To summarize, the SHA-256 algorithm is created with the aim of being highly secure against various types of attacks, such as preimage attacks, collision attacks, and birthday attacks. This makes it extremely difficult to find a message that produces a specific hash value, find two distinct messages with the same hash value, or find two messages that generate the same hash value. Therefore, it is valuable for ensuring the accuracy of data and identifying unauthorized modifications.

Figure 3.14: Elliptic Curves

3.5 ECDSA

3.5.1 Elliptic Curves

The Elliptic curve will be the set of points described by the equation

$$y^2 = x^3 + ax + b, \tag{3.1}$$

where a and b are constants and x and y are the coordinates of the points on the curve.

In Elliptic Curve Digital Signature Algorithm we deal with what is called Weierstrass normal form for elliptic curves with the equation

$$\left\{(x, y) \in \mathbb{R}^2 \mid y^2 = x^3 + ax + b,\ 4a^3 + 27b^2 \neq 0\right\} \cup \{0\} \tag{3.2}$$

Depending on the value of a and b, elliptic curves may assume different shapes on the plane and they are symmetric about the x-axis. The point at infinity will act as an ideal point for our curve.

3.5.2 Group Laws for Elliptic Curves

- The elements of the group are the points of an elliptic curve;

- The identity element is the point at infinity 0;
- The inverse of a point P is the one symmetric about the x-axis;
- Addition is given by the following rule: given three aligned, non-zero points P, Q and R, their sum is $P + Q + R = 0$.

Note that this definition assumes that the elliptic curve is defined over a field that supports the arithmetic operations necessary for the group law.

Geometric Addition

In the context of elliptic curve cryptography, the sum between two points P and Q can be computed using a geometric method. The formula $P + Q + R = 0$ can be written as $P + Q = -R$. To find the point R, one can draw a line passing through P and Q, which will intersect a third point on the curve, denoted by R. This intersection is implied by the fact that P, Q, and R are aligned. The required result of $P + Q$ can then be obtained by taking the inverse of R, denoted by $-R$. [8]

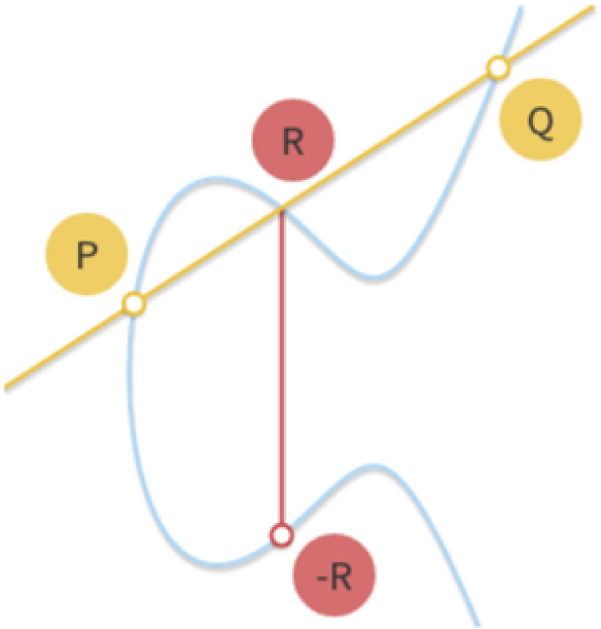


Figure 3.15: Geometric Addition

Algebraic Addition

1. Two non-zero, non-symmetric points $P = (x_p, y_p)$ and $Q = (x_q, y_q)$ are considered. If P and Q are distinct $(x_p \neq x_q)$, the slope of the line passing

through them is given by

$$m = (y_p - y_q)/(x_p - x_q) \tag{3.3}$$

2. The third point of intersection of this line with the elliptic curve is denoted by $R = (x_r, y_r)$ and can be determined using the following equations:

$$x_r = m^2 - x_p - x_q$$

$$y_r = y_p + m(x_r - x_p) \quad \text{or} \quad y_r = y_q + m(x_r - x_q)$$

3. Therefore, the sum of the two points (x_p, y_p) and (x_q, y_q) is given by $(x_r, -y_r)$ (the signs need to be taken into account and it should be remembered that $P + Q = -R$). The validity of this result can be verified by checking whether R belongs to the curve and whether P, Q, and R are aligned.[9]

Point Doubling

ECDSA, point doubling refers to the process of adding a point to itself algebraically. Let $P = (x_p, y_p)$ be a non-zero point on an elliptic curve E represented in Weierstrass form. The doubling of P can be computed as follows:

1. Calculate the slope m of the tangent line to E at point P using the following formula:

$$m = \frac{3x_p^2 + a}{2y_p} \tag{3.4}$$

where a is the coefficient of the x^3 term in the Weierstrass equation for E.

2. Calculate the x and y coordinates of the point $2P$ using the following formulas:

$$x_{2P} = m^2 - 2x_p$$

$$y_{2P} = m(x_p - x_{2P}) - y_p$$

or equivalently,

$$y_{2P} = -y_p - m(x_{2P} - x_p)$$

The resulting point $2P = (x_{2P}, y_{2P})$ is the doubling of P on the elliptic curve E. Note that the point at infinity does not have a doubling operation.

Scalar Multiplication

Scalar multiplication is the operation of multiplying a point on the elliptic curve by an integer scalar. Let P be a point on the curve, and k be a scalar. Then the scalar multiplication kP is defined as adding P to itself k times: [10]

$$kP = \underbrace{P + P + \cdots + P}_{k \text{ times}}$$

This can be done using repeated point addition, but there are more efficient algorithms such as the double-and-add algorithm and the Montgomery ladder algorithm. These algorithms are based on the binary representation of the scalar k.

The double-and-add algorithm works as follows: Starting with $Q_0 = \mathcal{O}$ (the point at infinity), compute Q_i as follows:

1. If the i-th bit of k is 1, then set $Q_i = Q_{i-1} + P$.
2. Otherwise, set $Q_i = 2Q_{i-1}$ (i.e., double the previous point).

After n iterations, $Q_n = kP$. This algorithm requires at most $2\log_2(k)$ point additions and $log_2(k)$ point doublings.

3.5.3 Elliptic Curves in a Finite Field

An elliptic curve over a finite field $\mathbb{F}_p$ is defined by an equation of the form

$$E : y^2 \equiv x^3 + ax + b \pmod{p}, \tag{3.5}$$

where $a, b \in \mathbb{F}_p$ and $4a^3 + 27b^2 \neq 0 \pmod{p}$. The points on the curve E form a group denoted by $E(\mathbb{F}_p)$.

Domain Parameters of an Elliptic Curve over a Finite Field

An elliptic curve over a finite field is defined by six domain parameters, which are as follows:

- A prime field F_p where p is a large prime number.
- A coefficient a and b from the field F_p such that the elliptic curve equation E is defined by $y^2 = x^3 + ax + b$.
- A point P on the elliptic curve with coordinates (x_P, y_P), which generates a cyclic subgroup of prime order n.
- The order n of the cyclic subgroup generated by the point P.

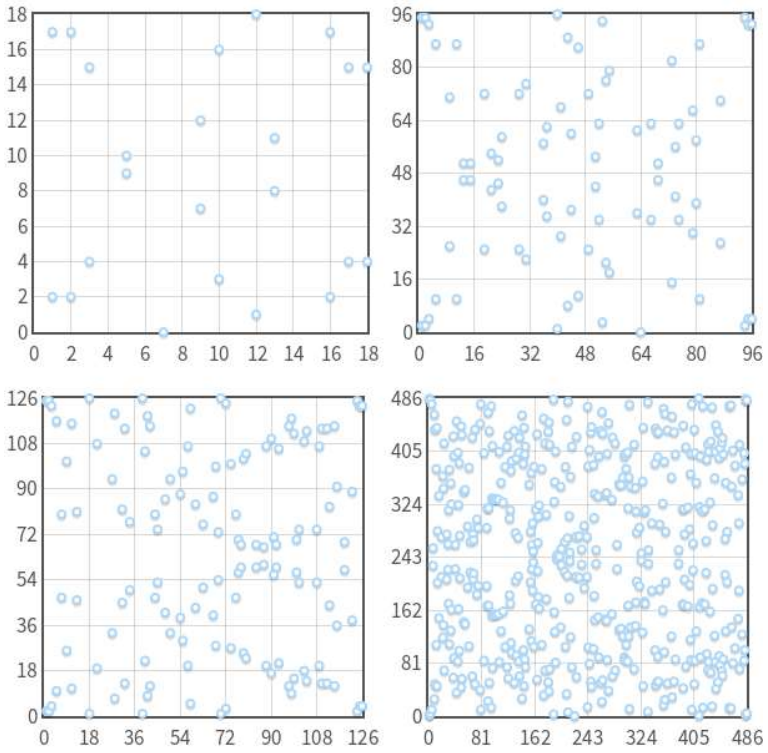


Figure 3.16: Elliptic Curves in Finite Fields

- The cofactor h, which is defined as the number of points on the elliptic curve divided by the order of the cyclic subgroup generated by P, i.e., $h = \#E(F_p)/n$.

The domain parameters are usually chosen in such a way that the order n of the cyclic subgroup generated by the point P is a large prime number. This is important for the security of cryptographic schemes that use elliptic curves, such as ECDSA.

The elliptic curve equation E over F_p can be written as:

$$E : y^2 \equiv x^3 + ax + b \pmod{p}*$$

where $\equiv$ denotes congruence modulo p.

The point P on the elliptic curve is chosen such that its order n is prime, i.e., $nP = \mathcal{O}$ where $\mathcal{O}$ is the point at infinity. The order n is also used to define a cyclic subgroup G of E, which is generated by P and has n elements:

$$G = \langle P \rangle = \{\mathcal{O}, P, 2P, 3P, \ldots, (n-1)P\}$$

The cofactor h is defined as the number of points on the elliptic curve divided by the order of the cyclic subgroup generated by P:

$$h = \frac{\#E(F_p)}{n}$$

where $\#E(F_p)$ denotes the number of points on the elliptic curve over F_p.

In order to ensure the security of cryptographic schemes based on elliptic curves, it is important to choose the domain parameters carefully and to use parameters that have been standardized and thoroughly vetted by the cryptographic community.

The group arithmetic defined for Elliptic Curves are still valid in the finite fields

The elliptic curve used in this project is the one used for signature in Bitcoins which is secp256k1. Curve parameters are as follows

1. `secp256k1`
2. Field characteristic: p = 0xFFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F
3. Curve coefficients: $a = 0, b = 7$
4. Base point: g = (0x79be667e f9dcbbac 55a06295 ce870b07 029bfcdb 2dce28d9 59f2815b 16f81798 0x 483ada77 26a3c465 5da4fbfc 0e1108a8 fd17b448 a6855419 9c47d08f fb10d4b8)
5. Subgroup order: n = 0x FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141
6. Subgroup cofactor: h=1

3.5.4 Design Architecture

The design has several sub-modules which are integrated together to achieve the functionality. There is a custom ALU for Modular mathematics used in the finite field Elliptic Curve operations, and algebraic point arithmetics using modules that help find slopes between two points and tangents and followed by another module that does the resultant point calculation with the obtained slope and given points.

ALU

The Verilog-designed ECDSA module utilizes a custom ALU for performing mathematical operations required for Elliptic Curve Cryptography. This ALU is built using basic arithmetic and logic functions such as addition, subtraction, multiplication, division, modulus, and comparison. Additionally, a modular arithmetic module has been developed using these functions as building blocks, specifically for performing arithmetic operations like addition, multiplication, subtraction, and division on a finite field.

Inverse module function is also an essential math in finite field operations and Extended Euclidean Algorithm is used to obtain the value for this which is explained below

1. Given two integers a and m, where m is prime, set $b = m$ and $x = 0$ and $y = 1$.

2. Using the standard Euclidean algorithm, compute the greatest common divisor of a and m, along with the quotients $q_0, q_1, q_2, ...$ and remainders $r_0, r_1, r_2, ...$

3. Write the greatest common divisor as a linear combination of a and m: $d = as + mt$, where d is the greatest common divisor and s and t are integers.

4. If $d \neq 1$, then a is not invertible modulo m. Otherwise, the inverse of a modulo m is given by s.

With this put together the ECDSA ALU does the following operations

1. Modular Addition

2. Modular Subtraction

3. Modular Multiplication

4. Modular Division

5. Inverse Modulo

Slope Calculation

The module calculates the slope for point addition or doubling and passes it to the next module. The pseudo-code explains these steps

if $P = Q$ **then**
    $m \leftarrow (3x_1^2 + a)/(2y_1)$ ▷ Calculate tangent slope
else
    $m \leftarrow (y_2 - y_1)/(x_2 - x_1)$ ▷ Calculate slope between two points
end if
Output m ▷ Output the slope value

Point Calculation

This modules calculates the new point $R(x, y)$ from point $P(x, y) and Q(x, y)$ and the slope m obtained from the slope calculation module. The pseudo-code that corresponds to this step is as follows

1: $x \leftarrow m^2 - P.x - Q.x$
2: $y \leftarrow m \cdot (P.x - x) - P.y$
3: $R.x \leftarrow x$
4: $R.y \leftarrow y$

Scalar Multiplication

This module does the Double and Add Algorithm to calculate the scalar product of a given point. It uses both the slope calculation and point calculation modules to obtain the scalar product. The pseudo-code given below explains the working

Require: k is a scalar, P is a point on an elliptic curve E
Ensure: kP

function DOUBLEANDADDSLOPE(k, P)
    $Accum \leftarrow \mathcal{O}$ ▷ Initialize $Accum$ to the point at infinity
    $k_{bin} \leftarrow$ binary representation of k
    for $i \leftarrow 0$ to $\text{length}(k_{bin}) - 1$ **do**
        if $k_{bin}[i] = 1$ **then**
            $slope \leftarrow$ slope of line through P and $Accum$ ▷ Calculate the slope
            $Q \leftarrow$ point resulting from adding P and $Accum$ with slope $slope$ ▷ Calculate the new point
            $Accum \leftarrow Q$ ▷ Set the new point as the current accumulator
        end if
        $P \leftarrow$ double of P using slope $slope$ ▷ Double P using the previous slope
    end for
    return $Accum$
end function

Main Block

This is the top module that integrates all other components. This module generates the digital signal. One instance of the ALU module is used and a common is bus used to connect it to other components of the main block and is also used by the main block as well. The following pseudo-code explains the process in brief.

Require: G is the generator point, d is the private key, p and n are curve parameters, H is the hash function, k is a random value
Ensure: ECDSA signature (r, s)

function ECDSASIGN(G, d, p, n, H, k)
    $kG \leftarrow$ scalar product of k and G ▷ Compute the scalar product of k and G
    $r \leftarrow kG.x \bmod n$ ▷ Compute the x coordinate of the scalar product and take mod n
    $e \leftarrow H(m)$ ▷ Compute the hash value of the message
    $s \leftarrow (k^{-1}(e + dr)) \bmod n$ ▷ Compute the signature component s
    return (r, s)

In this algorithm, the scalar product of the random value k and the generator point G is computed to obtain a point on the elliptic curve. The x coordinate of this point is then taken mod n to obtain the r component of the signature. The hash value of the message is computed using the specified hash function H, and this value is denoted as e. Finally, the s component of the signature is computed using the formula $(k^{-1}(e + dr)) \bmod n$. Note that d is the private key and is used in the computation of s.

CHAPTER 4

Results and Discussions

4.1 Simulation

Simulation is the process of creating a computer model of a real-world system or process and using that model to investigate or study the behavior and performance of the system under different scenarios or conditions. It is a technique that allows researchers, engineers, scientists, and other professionals to test, analyze, and optimize complex systems or processes without the need for expensive and time-consuming physical experiments.

Simulations can be used in various fields, such as engineering, physics, economics, biology, and social sciences, among others. They can range from simple models that involve only a few variables to highly complex models that incorporate many interrelated factors. Simulations can help to identify and address problems or issues, optimize performance, predict outcomes, and support decision-making processes.

4.1.1 AES-GCM

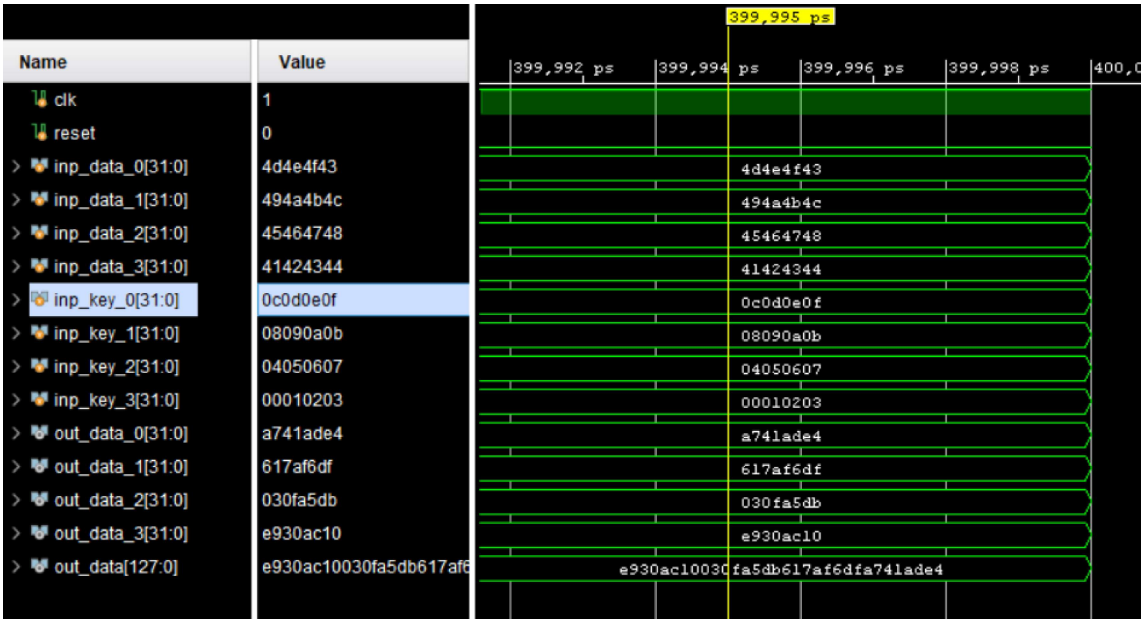


Figure 4.1: AES Simulation Result

Here the input data to the AES Algorithm is 128 bits and in hexadecimal its represented as ”414243444546474849 4a4b4c4d4e4f43”. The AES key used for simulation is also of 128 bits and in hexadecimal its represented as ”000102030405060708090a0b0c0d0e0f”. The encrypted data in hexadecimal is ”e930ac10030fa5db617af6dfa741ade4”.

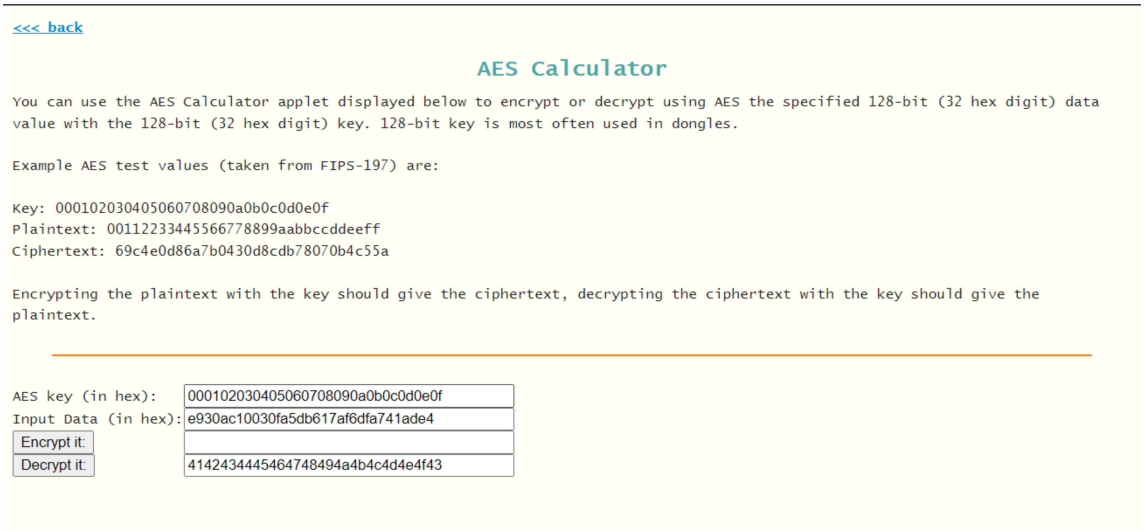


Figure 4.2: AES Verification

Here we have used the same input AES key that was used for simulation and the input data is encrypted data we obtained after AES Simulation. The decrypted output in hexadecimal is ”414243444546474849 4a4b4c4d4e4f43” which is the input we gave to our AES algorithm for simulation. Hence its successfully verified.

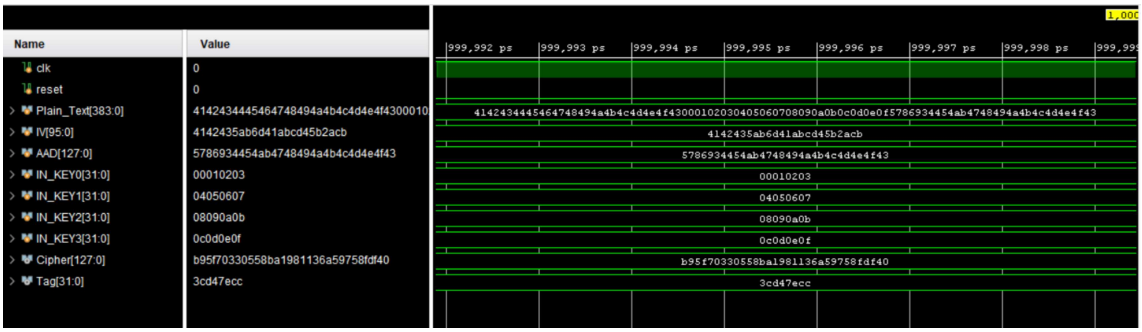


Figure 4.3: AES-GCM Simulation

Here input Plaintext is 384 bits and in hexadecimal its represented as 414243444 5464748494a4b4c4d4e4f43000102030405060708090a0b0c0d0e0f5786934454ab474 8494a4b4c4d4e4f43. The Initialization Vector is of 96 bits and in hexadecimal its represented as ”4142435ab6d41abcd45b2acb”. The AES key used is ”000102030405 060708090a0b0c0d0e0f”. The Additional Authenticated data is of 128 bits and in hexadecimal its

”5786934454ab4748494a4b4c4d4e4f43”. The final encrypted output after simulation is of 384 bits and in hexadecimal its represented as ”d20d9dbba42180babe2fb56bff45 afcc858a0cfcf48fc1f5ff6cf42cbe06ee80c4c94dbbb5cc80babe2fb56bff45afcc”.

4.1.2 SHA & ECDSA

The input data

ID (32 Bits) – 0x9a35b0e2

Voltage (64 Bits) – 0x35bd74c41e98b44
Mag - 219.841 (0x435bd74c)
Pha - 29.193 (0x41e98b44)

Current (64 Bits) – 0x40c6d91742187dd9
Mag - 6.214 A (0x40c6d917)
Pha - 8.1229 (0x42187dd9)

Power (64 Bits) - 0x00000000449a1bf8 (1232.874 W)

Energy (64 Bits) – 0x0000000042707bb3 (60.1208 kwh)

Date and Time
Day (8 Bits) - 0x08
Month (8 Bits) - 0x05
Year (16 bit) - 0x07E7
Hours (8 Bits) - 0x0a
Minutes (12 Bits) - 0x012
Seconds (12 Bits) - 0x021

Floating point harmonic distortion (32 bit) 2.1045 V (0x4006b021)

Frequency (32 Bits) 00000032

SHA input data Block
0x9a35b0e2435bd74c41e98b4440c6d91742187dd900000000449a1bf80000000
042707bb3080507e70a0120214006b021000000328000000000000000000001a0

Hash Value
0x5e7474c0fa6142ebfd3da11e7d571755ae42ba3860faa41ccc589df1d9b2e88c

Digital Signature R
0x2b4ea0a797a443d293ef5cff444f4979f06acfebd7e86d277475656138385b6c

Digital Signature S
0x4fa4cb26df088990aac824f4b91945104f5f9b701c41bcc61c0fa7fd8bd42576

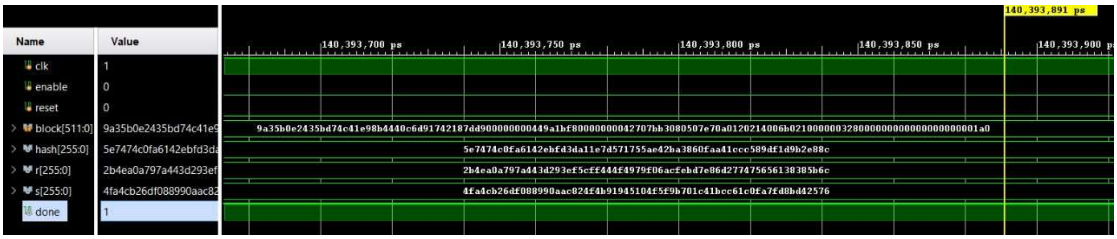


Figure 4.4: SHA ECDSA Simulations

```
[Running] python -u "e:\College\Internship and Projects\Digital\Smart Meter\ecdsa_py.py"
Curve: secp256k1
Private key: 0x2ab
Public key: (0xc0613eb61d6755eb2ebc9284a0aa69d288c390504b7e869ff3d943aa067ea65d, 0x72f61ae527b5c82e4afa896655d9289d29931c1bf0d36c09fe4213c527848cba)
Hash :  5e7474c0fa6142ebfd3da11e7d571755ae42ba3860faa41ccc589df1d9b2e88c

Message: 9a35b0e2435bd74c41e98b4440c6d91742187dd900000000449a1bf80000000042707bb3080507e70a0120214006b02100000032
Signature: (0x2b4ea0a797a443d293ef5cff444f4979f06acfebd7e86d277475656138385b6c, 0x4fa4cb26df088990aac824f4b91945104f5f9b701c41bcc61c0fa7fd8bd42576)
Hash :  5e7474c0fa6142ebfd3da11e7d571755ae42ba3860faa41ccc589df1d9b2e88c
Verification: signature matches
```

Figure 4.5: SHA ECDSA Verification

Upon observing the provided figure, it can be noted that the input block of the SHA undergoes processing within the SHA module, resulting in the generation of a hash value. This hash value is then passed onto the ECDSA module, where the digital signature is created by producing the r and s values. It is noteworthy that the generated hash value from the hardware simulation corresponds to the hash value generated from the Python script. This observation leads to the conclusion that the modules' functionalities are logically correct, as expected.

4.2 Synthesis

Synthesis refers to the process of transforming a high-level hardware description language (HDL) code, such as Verilog or VHDL, into a gate-level netlist that can be used to configure an FPGA device. The synthesis process involves analyzing the HDL code and translating it into a logic-level representation of the design. During synthesis, the HDL code is analyzed and optimized to create a gate-level netlist that is functionally equivalent to the original design. This netlist is then used to configure

the FPGA device, which consists of programmable logic blocks, interconnects, and I/O blocks that can be configured to implement any digital logic function.

4.2.1 AES-GCM

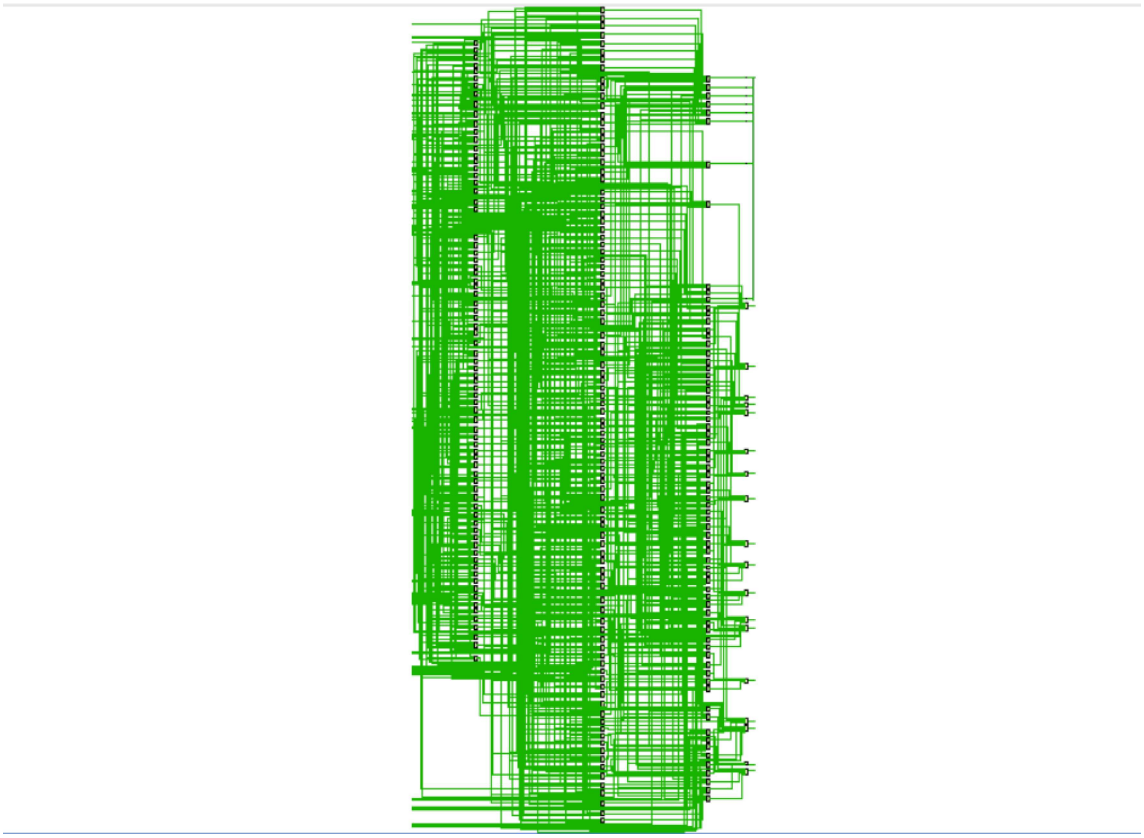


Figure 4.6: AES-GCM RTL Schematic

Name	Slice LUTs (20800)	Bonded IOB (106)
mult_in_GF128	7715	384

Figure 4.7: AES-GCM Utilization

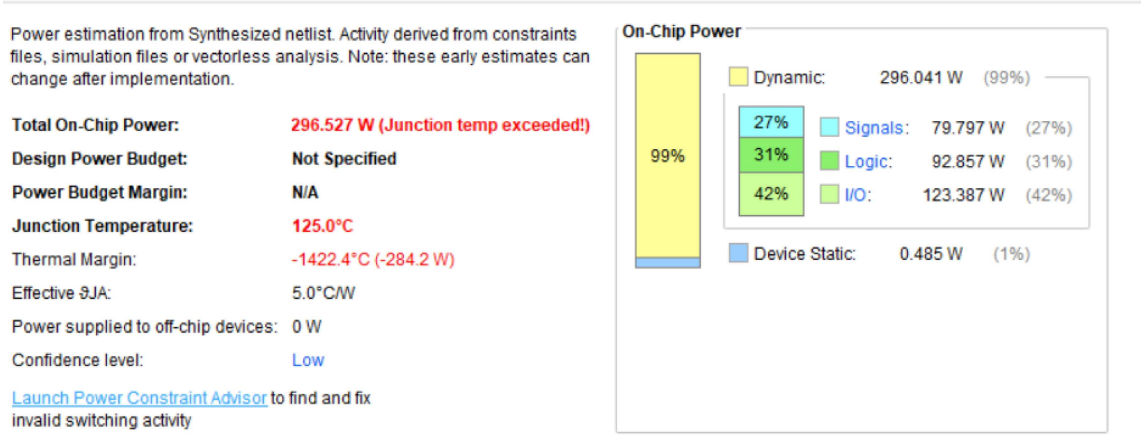
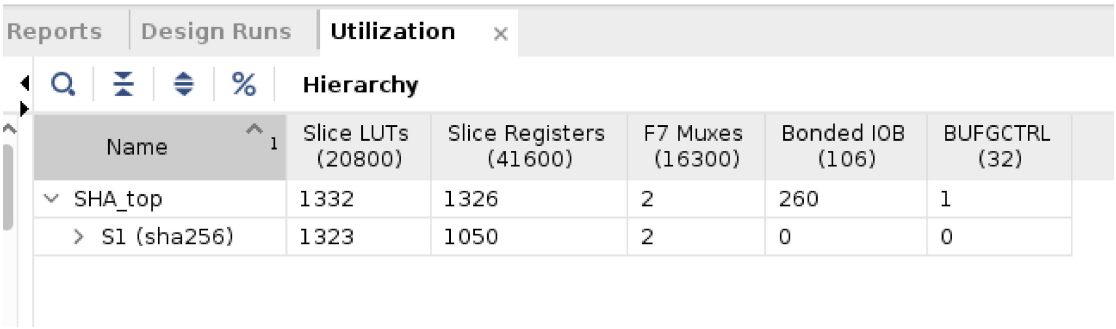


Figure 4.8: AES-GCM Power

We synthesized the AES-GCM algorithm and generated the RTL schematic. The schematic consists of 9469 nets and 9213 leaf cells. From fig 4.5 it can be observed that total LUTs used is 7715. From fig 4.6 we can observe that static power dissipation is 0.485 W and dynamic power dissipation is 296.041 W. The total on chip power required is 296.527 W.

4.2.2 SHA



Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	Bonded IOB (106)	BUFGCTRL (32)
SHA_top	1332	1326	2	260	1
S1 (sha256)	1323	1050	2	0	0

Figure 4.9: SHA256 Utilization Report

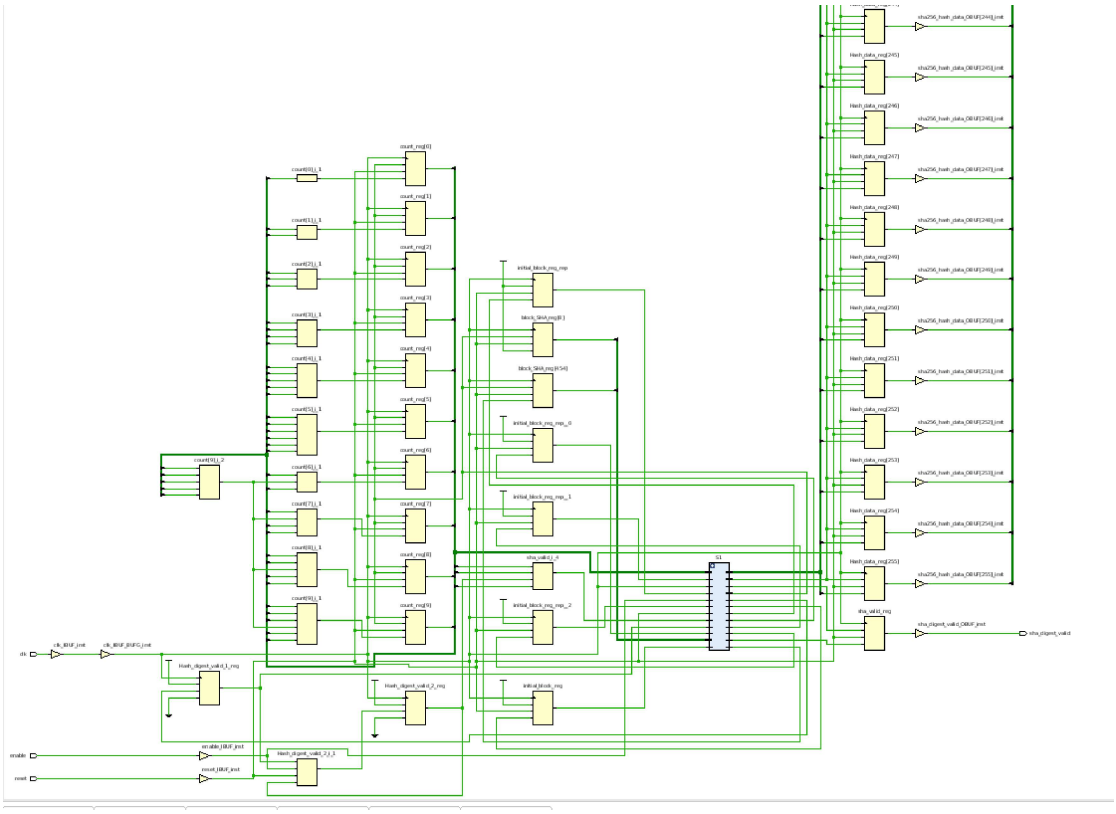


Figure 4.10: SHA256 Schematics

Based on an analysis of the utilization report and schematics, it can be inferred that the hardware design of SHA256 has been successfully implemented. The SHA256 module employs 1332 LUTs and 1326 slice registers, which are well within the available limit of FPGA LUTs and slice registers.

However, it is noteworthy that the Bonded IOB pins have exceeded the limit. This can be attributed to the fact that the SHA256 module was synthesized without the Encryption_top module, which is responsible for transmitting the hash data through a TxD line in the UART communication protocol.

4.2.3 ECDSA

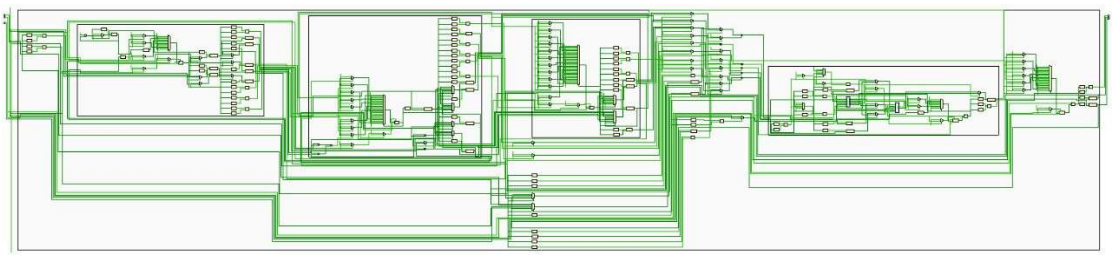


Figure 4.11: ECDSA Schematics

Based on the analysis of schematics, it can be inferred that the hardware design of ECDSA has been successfully implemented.

4.3 Implementation

The feasibility of the hardware design was tested by implementing the hardware designs on the Basys 3 board. This was done to ensure that the design could be effectively converted into an electronic circuit. In the design, separate registers were used to store the input data that would be processed by the encryption algorithm of each module. This approach helped to reduce the need for external inputs to the FPGA, thereby minimizing the work required for communication between the FPGA peripherals and the registers.

To facilitate the transmission and reading of output data from the FPGA, the outputs of each algorithm were also stored in registers. The UART communication protocol was used to read the output data from the FPGA, and a USB RS232 connection was employed to connect the FPGA to the computing machine. This approach ensured that the output data could be easily transmitted and read.

The implementation of the design on the Basys 3 board helped to confirm the feasibility of the hardware design. It also provided valuable insights into the performance and functionality of the design, which can be used to further optimize and improve the design. Overall, the use of separate registers for input data storage proved to be an effective approach, which helped to simplify the communication process and reduce the workload required for data transfer.

4.3.1 AES-GCM

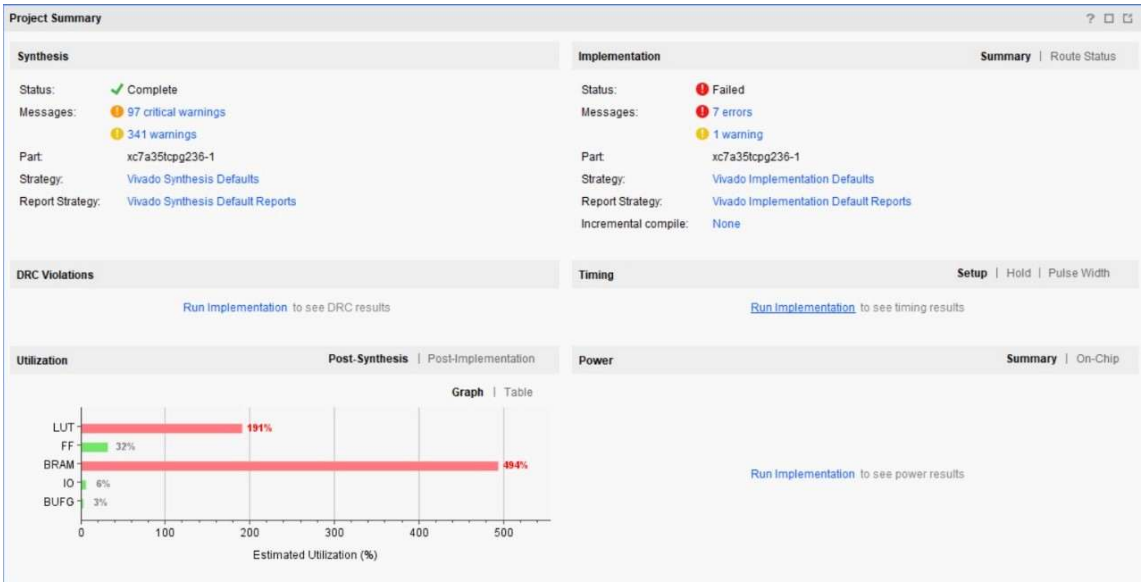


Figure 4.12: AES-GCM FPGA

The FPGA board which was used for implementation did not have enough registers to execute the algorithm hence bitstream generation was not possible. As observed from the figure LUT and BRAM ran out of space to store the algorithm.

4.3.2 SHA

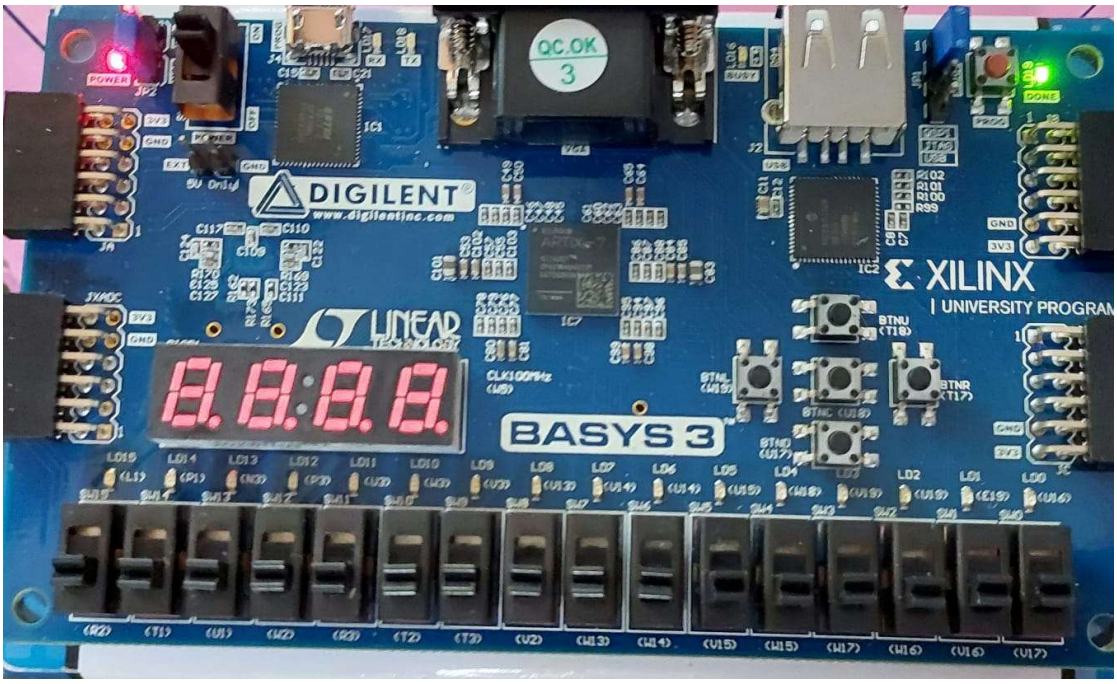


Figure 4.13: SHA256 FPGA implementation

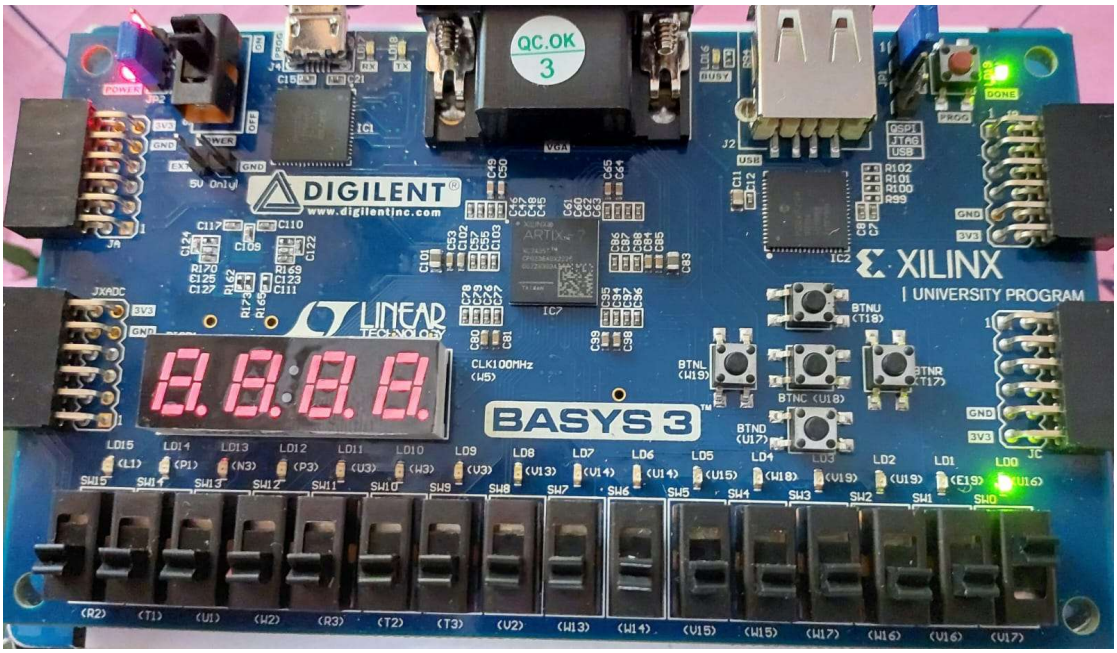


Figure 4.14: SHA256 FPGA Enable signal

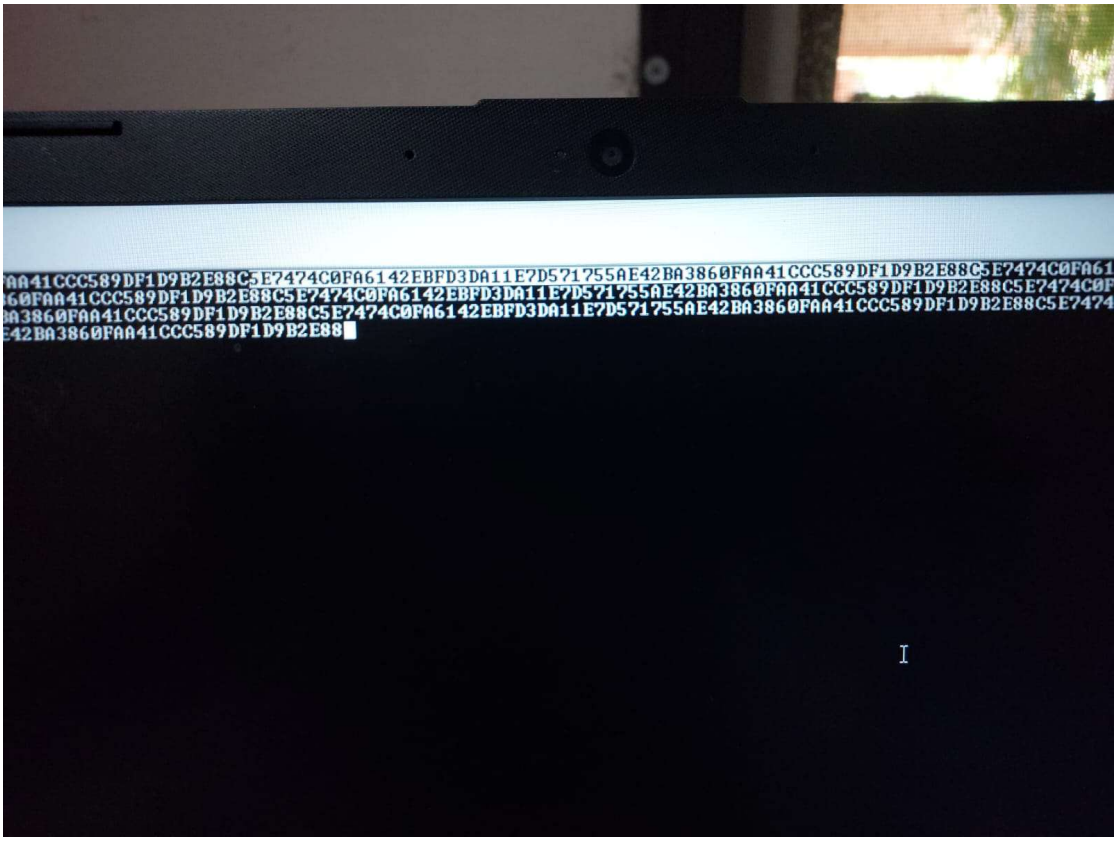


Figure 4.15: SHA256 UART Output

After analyzing the provided figures, it can be observed that the FPGA has been programmed with SHA256. In the first figure, the Led-U16 and switch-V16 are both turned off, indicating that the enable signal has not been asserted using the V16

switch. However, in the second figure, the switch-V16 is turned on, and the Led-U16 is also turned on, indicating that the SHA module is performing the hashing process. The module completes the hashing process by asserting Hash_digest_valid, which is represented by the Led-U16 turning on.

Moreover, the result obtained from the hashing process is stored in the register and can be read out from the FPGA using the UART protocol. It is noteworthy that the output of the Hash value matches the simulation-verified output, confirming that the hardware design is functioning correctly.

In summary, the analysis of the figures suggests that the FPGA has been successfully programmed with SHA256. The activation of the Led-U16 and switch-V16 confirms that the enable signal has been asserted, allowing the SHA module to perform the hashing process. The verification of the output using simulation adds to the confidence in the hardware design's effectiveness. Overall, these observations provide assurance in the accuracy and reliability of the SHA256 hardware design.

CHAPTER 5

Conclusion

5.1 Summary

The FPGA implementation of level 1 security suite of DLMS protocol for secure transmission of data in smart meters is discussed in this paper. The level 1 security suite of DLMS consists of AES-GCM for encryption of plain text, SHA-256 for generation of hash, ECDH for key agreement and key exchange, and ECDSA for generation of digital signature. We have considered that the key exchange between the sender and receiever has already been done using the ECDH algorithm.

Based on results from Chapter 4, the level of implementation of the security suite is as follows :

1. SHA-256 algorithm was successfully simulated in Xilinx Vivado and verified using online tools. We were able to synthesize the algorithm and generate the bitstream for the Basys 3 FPGA board.

2. AES-GCM algorithm was successfully simulated in Xilinx Vivado. AES is one of main modules present inside this algorithm which was seperately simulated and verified using online tools. We were able to synthesize the algorithm but could not generate the bitstream for Basys 3 FPGA board as it did not have enough memory space.

3. ECDSA algorithm was successfully simulated in Xilinx Vivado and verified using a python script.

5.2 Future Scope

This project has several potential future scopes that can further enhance the security of data transmission in smart meters. These include:

1. Integration with other communication protocols: The project can be extended to include other communication protocols used in smart grid technology, such as Modbus and DNP3, to provide secure communication across various devices.

2. Implementation of other encryption algorithms: The project can incorporate other encryption algorithms, such as RSA (Rivest–Shamir–Adleman), to provide a wider range of secure communication options.

3. Development of a comprehensive security framework: The project can serve as a foundation for developing a comprehensive security framework for smart grid technology, addressing various security concerns across different layers of the system.

4. Integration with machine learning algorithms: The project can be combined with machine learning algorithms to enhance the security and privacy of smart meters by detecting anomalies in energy usage data and preventing cyber attacks.

5. Development of a user-friendly interface: The project can be extended to develop a user-friendly interface for smart meter users, providing them with an easy way to verify the security of their data transmission.

REFERENCES

[1] Jixuan Zheng, David Wenzhong Gao, and Li Lin. “Smart Meters in Smart Grid: An Overview”. In: (2013), pp. 57–64.

[2] Pin-Yu Chen, Shin-Ming Cheng, and Kwang-Cheng Chen. “Smart attacks in smart grid communication networks”. In: *IEEE Communications Magazine* 50.8 (2012), pp. 24–29.

[3] N Radhikaa Divya M Menona. “Design of a Secure Architecture for Last Mile Communication in Smart Grid Systems”. In: (2015).

[4] Tai-hoon Kim Maricel O Balitanas Rosslin John Robles. “Retrofit to CAIN Issues for Critical Infrastructures”. In: *Communications in Computer and Information Science book series* 223 ().

[5] Iskraemeco d.d. Győző Kmethy DLMS UA President Emeritus Gnarus Engineering Ltd Milan Kozole Convenor of the DLMS UA Technical Standing Committee. “Security in DLMS”. In: *A White Paper by the DLMS User Association* (2019).

[6] Morris Dworkin. “Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC”. In: (2007).

[7] John Viega David A. McGrew. “The Galois/Counter Mode of Operation (GCM)”. In: ().

[8] ANDREA CORBELLINI. “Elliptic Curve Cryptography: ECDH and ECDSA”. In: *ECC: a gentle introduction.* 3 (2015).

[9] ANDREA CORBELLINI. “Elliptic Curve Cryptography: finite fields and discrete logarithms”. In: *ECC: a gentle introduction* 1 (2015).

[10] ANDREA CORBELLINI. “Elliptic Curve Cryptography: a gentle introduction”. In: *ECC: a gentle introduction.* 2 (2015).

APPENDIX A

CODE ATTACHMENTS

A.1 AES

A.1.1 TOP-MODULE

```
1  module AES128(
2      input clk,
3      input reset,
4
5      input [31:0] IN_DATA0,
6      input [31:0] IN_DATA1,
7      input [31:0] IN_DATA2,
8      input [31:0] IN_DATA3,
9
10     input [31:0] IN_KEY0,
11     input [31:0] IN_KEY1,
12     input [31:0] IN_KEY2,
13     input [31:0] IN_KEY3,
14
15     output [31:0] OUT_DATA0,
16     output [31:0] OUT_DATA1,
17     output [31:0] OUT_DATA2,
18     output [31:0] OUT_DATA3
19 );
20 wire [127:0] IN_DATA, IN_KEY, OUT_DATA;
21
22 assign IN_DATA = {IN_DATA3, IN_DATA2, IN_DATA1, IN_DATA0};
23 assign IN_KEY = {IN_KEY3, IN_KEY2, IN_KEY1, IN_KEY0};
24 assign {OUT_DATA3, OUT_DATA2, OUT_DATA1, OUT_DATA0} = OUT_DATA;
25
26 reg [127:0] R0_OUT_DATA, KEY;
27 wire [127:0] R1_OUT_DATA, R2_OUT_DATA, R3_OUT_DATA, R4_OUT_DATA,
       R5_OUT_DATA, R6_OUT_DATA, R7_OUT_DATA, R8_OUT_DATA, R9_OUT_DATA;
28 wire [127:0] OUT_KEYW1, OUT_KEYW2, OUT_KEYW3, OUT_KEYW4, OUT_KEYW5,
       OUT_KEYW6, OUT_KEYW7, OUT_KEYW8, OUT_KEYW9;
29 wire [127:0] OUT_KEYR0, OUT_KEYR1, OUT_KEYR2, OUT_KEYR3, OUT_KEYR4,
       OUT_KEYR5, OUT_KEYR6, OUT_KEYR7, OUT_KEYR8, OUT_KEYR9;
30
31 always @(posedge clk) begin
32     R0_OUT_DATA <= IN_DATA ^ IN_KEY;
33     KEY <= IN_KEY;
34 end
35
36 GENERATE_KEY
37         K1 (.clk(clk), .ROUND_KEY(4'd0), .IN_KEY(KEY),          .OUT_KEY
               (OUT_KEYW1), .OUT_KEY_R(OUT_KEYR0)),
38         K2 (.clk(clk), .ROUND_KEY(4'd1), .IN_KEY(OUT_KEYW1), .OUT_KEY
               (OUT_KEYW2), .OUT_KEY_R(OUT_KEYR1)),
```

```
39          K3  (.clk(clk), .ROUND_KEY(4'd2), .IN_KEY(OUT_KEYW2), .OUT_KEY
                (OUT_KEYW3), .OUT_KEY_R(OUT_KEYR2)),
40          K4  (.clk(clk), .ROUND_KEY(4'd3), .IN_KEY(OUT_KEYW3), .OUT_KEY
                (OUT_KEYW4), .OUT_KEY_R(OUT_KEYR3)),
41          K5  (.clk(clk), .ROUND_KEY(4'd4), .IN_KEY(OUT_KEYW4), .OUT_KEY
                (OUT_KEYW5), .OUT_KEY_R(OUT_KEYR4)),
42          K6  (.clk(clk), .ROUND_KEY(4'd5), .IN_KEY(OUT_KEYW5), .OUT_KEY
                (OUT_KEYW6), .OUT_KEY_R(OUT_KEYR5)),
43          K7  (.clk(clk), .ROUND_KEY(4'd6), .IN_KEY(OUT_KEYW6), .OUT_KEY
                (OUT_KEYW7), .OUT_KEY_R(OUT_KEYR6)),
44          K8  (.clk(clk), .ROUND_KEY(4'd7), .IN_KEY(OUT_KEYW7), .OUT_KEY
                (OUT_KEYW8), .OUT_KEY_R(OUT_KEYR7)),
45          K9  (.clk(clk), .ROUND_KEY(4'd8), .IN_KEY(OUT_KEYW8), .OUT_KEY
                (OUT_KEYW9), .OUT_KEY_R(OUT_KEYR8)),
46          K10(.clk(clk), .ROUND_KEY(4'd9), .IN_KEY(OUT_KEYW9), .OUT_KEY
                (), .OUT_KEY_R(OUT_KEYR9));
47
48  ROUND_ITERATION R1(.clk(clk),
49                     .IN_DATA(R0_OUT_DATA),
50                     .IN_KEY(OUT_KEYR0),
51                     .OUT_DATA(R1_OUT_DATA));
52
53  ROUND_ITERATION R2(.clk(clk),
54                     .IN_DATA(R1_OUT_DATA),
55                     .IN_KEY(OUT_KEYR1),
56                     .OUT_DATA(R2_OUT_DATA));
57
58  ROUND_ITERATION R3(.clk(clk),
59                     .IN_DATA(R2_OUT_DATA),
60                     .IN_KEY(OUT_KEYR2),
61                     .OUT_DATA(R3_OUT_DATA));
62
63  ROUND_ITERATION R4(.clk(clk),
64                     .IN_DATA(R3_OUT_DATA),
65                     .IN_KEY(OUT_KEYR3),
66                     .OUT_DATA(R4_OUT_DATA));
67
68  ROUND_ITERATION R5(.clk(clk),
69                     .IN_DATA(R4_OUT_DATA),
70                     .IN_KEY(OUT_KEYR4),
71                     .OUT_DATA(R5_OUT_DATA));
72
73  ROUND_ITERATION R6(.clk(clk),
74                     .IN_DATA(R5_OUT_DATA),
75                     .IN_KEY(OUT_KEYR5),
76                     .OUT_DATA(R6_OUT_DATA));
77
78  ROUND_ITERATION R7(.clk(clk),
79                     .IN_DATA(R6_OUT_DATA),
80                     .IN_KEY(OUT_KEYR6),
81                     .OUT_DATA(R7_OUT_DATA));
82
83  ROUND_ITERATION R8(.clk(clk),
84                     .IN_DATA(R7_OUT_DATA),
85                     .IN_KEY(OUT_KEYR7),
86                     .OUT_DATA(R8_OUT_DATA));
```

```
87
88 ROUND_ITERATION R9(.clk(clk),
89                    .IN_DATA(R8_OUT_DATA),
90                    .IN_KEY(OUT_KEYR8),
91                    .OUT_DATA(R9_OUT_DATA));
92
93 LAST_ROUND R10(.clk(clk),
94                .IN_DATA(R9_OUT_DATA),
95                .IN_KEY(OUT_KEYR9),
96                .OUT_DATA(OUT_DATA));
97
98
99 endmodule
```

A.1.2 SUB-BYTES

```
1  module SUB_BYTES(
2      input clk,
3      input [127:0] IN_DATA,
4      output [127:0] SB_DATA
5  );
6
7  reg [127:0] SB_DATA;
8  wire [127:0] SB_DATA_W;
9
10     FORWARD_SUBSTITUTION_BOX INST0(.clk(clk), .A(IN_DATA[127:120]), .
           C(SB_DATA_W[127:120]));
11     FORWARD_SUBSTITUTION_BOX INST1(.clk(clk), .A(IN_DATA[119:112]), .
           C(SB_DATA_W[119:112]));
12     FORWARD_SUBSTITUTION_BOX INST2(.clk(clk), .A(IN_DATA[111:104]), .
           C(SB_DATA_W[111:104]));
13     FORWARD_SUBSTITUTION_BOX INST3(.clk(clk), .A(IN_DATA[103:96]), .C
           (SB_DATA_W[103:96]));
14
15     FORWARD_SUBSTITUTION_BOX INST4(.clk(clk), .A(IN_DATA[95:88]), .C(
           SB_DATA_W[95:88]));
16     FORWARD_SUBSTITUTION_BOX INST5(.clk(clk), .A(IN_DATA[87:80]), .C(
           SB_DATA_W[87:80]));
17     FORWARD_SUBSTITUTION_BOX INST6(.clk(clk), .A(IN_DATA[79:72]), .C(
           SB_DATA_W[79:72]));
18     FORWARD_SUBSTITUTION_BOX INST7(.clk(clk), .A(IN_DATA[71:64]), .C(
           SB_DATA_W[71:64]));
19
20     FORWARD_SUBSTITUTION_BOX INST8(.clk(clk), .A(IN_DATA[63:56]), .C(
           SB_DATA_W[63:56]));
21     FORWARD_SUBSTITUTION_BOX INST9(.clk(clk), .A(IN_DATA[55:48]), .C(
           SB_DATA_W[55:48]));
22     FORWARD_SUBSTITUTION_BOX INST10(.clk(clk), .A(IN_DATA[47:40]), .C
           (SB_DATA_W[47:40]));
23     FORWARD_SUBSTITUTION_BOX INST11(.clk(clk), .A(IN_DATA[39:32]), .C
           (SB_DATA_W[39:32]));
24
25     FORWARD_SUBSTITUTION_BOX INST12(.clk(clk), .A(IN_DATA[31:24]), .C
           (SB_DATA_W[31:24]));
26     FORWARD_SUBSTITUTION_BOX INST13(.clk(clk), .A(IN_DATA[23:16]), .C
           (SB_DATA_W[23:16]));
```

```
27      FORWARD_SUBSTITUTION_BOX INST14(.clk(clk), .A(IN_DATA[15:8]), .C(
            SB_DATA_W[15:8]));
28      FORWARD_SUBSTITUTION_BOX INST15(.clk(clk), .A(IN_DATA[7:0]), .C(
            SB_DATA_W[7:0]));
29
30    always @(*) begin
31          SB_DATA <= SB_DATA_W;
32      end
33
34  endmodule
```

A.1.3 MIX-COLUMNS

```
1       module MIX_COLUMNS(
2       input clk,
3       input [127:0] IN_DATA,
4       output [127:0] MIXED_DATA
5   );
6
7   reg [127:0] MIXED_DATA_REG;
8
9     always @(*) begin
10      MIXED_DATA_REG[127:120] = MIXCOLUMN(IN_DATA[127:120],IN_DATA
            [119:112],IN_DATA[111:104],IN_DATA[103:96]);
11      MIXED_DATA_REG[119:112] = MIXCOLUMN(IN_DATA[119:112],IN_DATA
            [111:104],IN_DATA[103:96],IN_DATA[127:120]);
12      MIXED_DATA_REG[111:104] = MIXCOLUMN(IN_DATA[111:104],IN_DATA
            [103:96],IN_DATA[127:120],IN_DATA[119:112]);
13      MIXED_DATA_REG[103:96] = MIXCOLUMN(IN_DATA[103:96],IN_DATA
            [127:120],IN_DATA[119:112],IN_DATA[111:104]);
14
15      MIXED_DATA_REG[95:88] = MIXCOLUMN(IN_DATA[95:88],IN_DATA[87:80],
            IN_DATA[79:72],IN_DATA[71:64]);
16      MIXED_DATA_REG[87:80] = MIXCOLUMN(IN_DATA[87:80],IN_DATA[79:72],
            IN_DATA[71:64],IN_DATA[95:88]);
17      MIXED_DATA_REG[79:72] = MIXCOLUMN(IN_DATA[79:72],IN_DATA[71:64],
            IN_DATA[95:88],IN_DATA[87:80]);
18      MIXED_DATA_REG[71:64] = MIXCOLUMN(IN_DATA[71:64],IN_DATA[95:88],
            IN_DATA[87:80],IN_DATA[79:72]);
19
20      MIXED_DATA_REG[63:56] = MIXCOLUMN(IN_DATA[63:56],IN_DATA[55:48],
            IN_DATA[47:40],IN_DATA[39:32]);
21      MIXED_DATA_REG[55:48] = MIXCOLUMN(IN_DATA[55:48],IN_DATA[47:40],
            IN_DATA[39:32],IN_DATA[63:56]);
22      MIXED_DATA_REG[47:40] = MIXCOLUMN(IN_DATA[47:40],IN_DATA[39:32],
            IN_DATA[63:56],IN_DATA[55:48]);
23      MIXED_DATA_REG[39:32] = MIXCOLUMN(IN_DATA[39:32],IN_DATA[63:56],
            IN_DATA[55:48],IN_DATA[47:40]);
24
25      MIXED_DATA_REG[31:24] = MIXCOLUMN(IN_DATA[31:24],IN_DATA[23:16],
            IN_DATA[15:8],IN_DATA[7:0]);
26      MIXED_DATA_REG[23:16] = MIXCOLUMN(IN_DATA[23:16],IN_DATA[15:8],
            IN_DATA[7:0],IN_DATA[31:24]);
27      MIXED_DATA_REG[15:8] = MIXCOLUMN(IN_DATA[15:8],IN_DATA[7:0],
            IN_DATA[31:24],IN_DATA[23:16]);
28      MIXED_DATA_REG[7:0] = MIXCOLUMN(IN_DATA[7:0],IN_DATA[31:24],
```

```
        IN_DATA[23:16],IN_DATA[15:8]);
29  end
30
31  assign MIXED_DATA = MIXED_DATA_REG;
32
33  function [7:0] MIXCOLUMN;
34  input [7:0] IN1, IN2, IN3, IN4;
35      begin
36          MIXCOLUMN[7] = IN1[6]^IN2[6]^IN2[7]^IN3[7]^IN4[7];
37          MIXCOLUMN[6] = IN1[5]^IN2[5]^IN2[6]^IN3[6]^IN4[6];
38          MIXCOLUMN[5] = IN1[4]^IN2[4]^IN2[5]^IN3[5]^IN4[5];
39          MIXCOLUMN[4] = IN1[3]^IN1[7]^IN2[3]^IN2[4]^IN2[7]^IN3[4]^IN4
                [4];
40          MIXCOLUMN[3] = IN1[2]^IN1[7]^IN2[2]^IN2[3]^IN2[7]^IN3[3]^IN4
                [3];
41          MIXCOLUMN[2] = IN1[1]^IN2[1]^IN2[2]^IN3[2]^IN4[2];
42          MIXCOLUMN[1] = IN1[0]^IN1[7]^IN2[0]^IN2[1]^IN2[7]^IN3[1]^IN4
                [1];
43          MIXCOLUMN[0] = IN1[7]^IN2[7]^IN2[0]^IN3[0]^IN4[0];
44      end
45  endfunction
46
47  endmodule
```

A.1.4 SHIFT-ROWS

```
1       module SHIFT_ROWS(
2     input clk,
3     input [127:0] IN_DATA,
4     output [127:0] SHIFT_DATA
5   );
6
7   reg [127:0] SHIFT_DATA_REG = 128'b0;
8
9     always @(*) begin
10        SHIFT_DATA_REG[127:120] = IN_DATA[127:120];
11        SHIFT_DATA_REG[119:112] = IN_DATA[87:80];
12        SHIFT_DATA_REG[111:104] = IN_DATA[47:40];
13        SHIFT_DATA_REG[103:96] = IN_DATA[7:0];
14
15        SHIFT_DATA_REG[95:88] = IN_DATA[95:88];
16        SHIFT_DATA_REG[87:80] = IN_DATA[55:48];
17        SHIFT_DATA_REG[79:72] = IN_DATA[15:8];
18        SHIFT_DATA_REG[71:64] = IN_DATA[103:96];
19
20        SHIFT_DATA_REG[63:56] = IN_DATA[63:56];
21        SHIFT_DATA_REG[55:48] = IN_DATA[23:16];
22        SHIFT_DATA_REG[47:40] = IN_DATA[111:104];
23        SHIFT_DATA_REG[39:32] = IN_DATA[71:64];
24
25        SHIFT_DATA_REG[31:24] = IN_DATA[31:24];
26        SHIFT_DATA_REG[23:16] = IN_DATA[119:112];
27        SHIFT_DATA_REG[15:8] = IN_DATA[79:72];
28        SHIFT_DATA_REG[7:0] = IN_DATA[39:32];
29      end
30
```

```
31      assign SHIFT_DATA = SHIFT_DATA_REG;
32
33  endmodule
```

A.1.5 FORWARD-SUBSTITURION

```
1       module FORWARD_SUBSTITUTION_BOX(
2     input clk,
3     input [7:0] A,
4     output [7:0] C
5   );
6
7   reg [7:0] C_REG;
8
9     always @(posedge clk) begin
10      case (A)
11      8'h00: C_REG <= 8'h63;
12      8'h01: C_REG <= 8'h7c;
13      8'h02: C_REG <= 8'h77;
14      8'h03: C_REG <= 8'h7b;
15      8'h04: C_REG <= 8'hf2;
16      8'h05: C_REG <= 8'h6b;
17      8'h06: C_REG <= 8'h6f;
18      8'h07: C_REG <= 8'hc5;
19      8'h08: C_REG <= 8'h30;
20      8'h09: C_REG <= 8'h01;
21      8'h0a: C_REG <= 8'h67;
22      8'h0b: C_REG <= 8'h2b;
23      8'h0c: C_REG <= 8'hfe;
24      8'h0d: C_REG <= 8'hd7;
25      8'h0e: C_REG <= 8'hab;
26      8'h0f: C_REG <= 8'h76;
27      8'h10: C_REG <= 8'hca;
28      8'h11: C_REG <= 8'h82;
29      8'h12: C_REG <= 8'hc9;
30      8'h13: C_REG <= 8'h7d;
31      8'h14: C_REG <= 8'hfa;
32      8'h15: C_REG <= 8'h59;
33      8'h16: C_REG <= 8'h47;
34      8'h17: C_REG <= 8'hf0;
35      8'h18: C_REG <= 8'had;
36      8'h19: C_REG <= 8'hd4;
37      8'h1a: C_REG <= 8'ha2;
38      8'h1b: C_REG <= 8'haf;
39      8'h1c: C_REG <= 8'h9c;
40      8'h1d: C_REG <= 8'ha4;
41      8'h1e: C_REG <= 8'h72;
42      8'h1f: C_REG <= 8'hc0;
43      8'h20: C_REG <= 8'hb7;
44      8'h21: C_REG <= 8'hfd;
45      8'h22: C_REG <= 8'h93;
46      8'h23: C_REG <= 8'h26;
47      8'h24: C_REG <= 8'h36;
48      8'h25: C_REG <= 8'h3f;
49      8'h26: C_REG <= 8'hf7;
50      8'h27: C_REG <= 8'hcc;
```

```
51          8'h28: C_REG <= 8'h34;
52          8'h29: C_REG <= 8'ha5;
53          8'h2a: C_REG <= 8'he5;
54          8'h2b: C_REG <= 8'hf1;
55          8'h2c: C_REG <= 8'h71;
56          8'h2d: C_REG <= 8'hd8;
57          8'h2e: C_REG <= 8'h31;
58          8'h2f: C_REG <= 8'h15;
59          8'h30: C_REG <= 8'h04;
60          8'h31: C_REG <= 8'hc7;
61          8'h32: C_REG <= 8'h23;
62          8'h33: C_REG <= 8'hc3;
63          8'h34: C_REG <= 8'h18;
64          8'h35: C_REG <= 8'h96;
65          8'h36: C_REG <= 8'h05;
66          8'h37: C_REG <= 8'h9a;
67          8'h38: C_REG <= 8'h07;
68          8'h39: C_REG <= 8'h12;
69          8'h3a: C_REG <= 8'h80;
70          8'h3b: C_REG <= 8'he2;
71          8'h3c: C_REG <= 8'heb;
72          8'h3d: C_REG <= 8'h27;
73          8'h3e: C_REG <= 8'hb2;
74          8'h3f: C_REG <= 8'h75;
75          8'h40: C_REG <= 8'h09;
76          8'h41: C_REG <= 8'h83;
77          8'h42: C_REG <= 8'h2c;
78          8'h43: C_REG <= 8'h1a;
79          8'h44: C_REG <= 8'h1b;
80          8'h45: C_REG <= 8'h6e;
81          8'h46: C_REG <= 8'h5a;
82          8'h47: C_REG <= 8'ha0;
83          8'h48: C_REG <= 8'h52;
84          8'h49: C_REG <= 8'h3b;
85          8'h4a: C_REG <= 8'hd6;
86          8'h4b: C_REG <= 8'hb3;
87          8'h4c: C_REG <= 8'h29;
88          8'h4d: C_REG <= 8'he3;
89          8'h4e: C_REG <= 8'h2f;
90          8'h4f: C_REG <= 8'h84;
91          8'h50: C_REG <= 8'h53;
92          8'h51: C_REG <= 8'hd1;
93          8'h52: C_REG <= 8'h00;
94          8'h53: C_REG <= 8'hed;
95          8'h54: C_REG <= 8'h20;
96          8'h55: C_REG <= 8'hfc;
97          8'h56: C_REG <= 8'hb1;
98          8'h57: C_REG <= 8'h5b;
99          8'h58: C_REG <= 8'h6a;
100         8'h59: C_REG <= 8'hcb;
101         8'h5a: C_REG <= 8'hbe;
102         8'h5b: C_REG <= 8'h39;
103         8'h5c: C_REG <= 8'h4a;
104         8'h5d: C_REG <= 8'h4c;
105         8'h5e: C_REG <= 8'h58;
106         8'h5f: C_REG <= 8'hcf;
```

```
107         8'h60: C_REG <= 8'hd0;
108         8'h61: C_REG <= 8'hef;
109         8'h62: C_REG <= 8'haa;
110         8'h63: C_REG <= 8'hfb;
111         8'h64: C_REG <= 8'h43;
112         8'h65: C_REG <= 8'h4d;
113         8'h66: C_REG <= 8'h33;
114         8'h67: C_REG <= 8'h85;
115         8'h68: C_REG <= 8'h45;
116         8'h69: C_REG <= 8'hf9;
117         8'h6a: C_REG <= 8'h02;
118         8'h6b: C_REG <= 8'h7f;
119         8'h6c: C_REG <= 8'h50;
120         8'h6d: C_REG <= 8'h3c;
121         8'h6e: C_REG <= 8'h9f;
122         8'h6f: C_REG <= 8'ha8;
123         8'h70: C_REG <= 8'h51;
124         8'h71: C_REG <= 8'ha3;
125         8'h72: C_REG <= 8'h40;
126         8'h73: C_REG <= 8'h8f;
127         8'h74: C_REG <= 8'h92;
128         8'h75: C_REG <= 8'h9d;
129         8'h76: C_REG <= 8'h38;
130         8'h77: C_REG <= 8'hf5;
131         8'h78: C_REG <= 8'hbc;
132         8'h79: C_REG <= 8'hb6;
133         8'h7a: C_REG <= 8'hda;
134         8'h7b: C_REG <= 8'h21;
135         8'h7c: C_REG <= 8'h10;
136         8'h7d: C_REG <= 8'hff;
137         8'h7e: C_REG <= 8'hf3;
138         8'h7f: C_REG <= 8'hd2;
139         8'h80: C_REG <= 8'hcd;
140         8'h81: C_REG <= 8'h0c;
141         8'h82: C_REG <= 8'h13;
142         8'h83: C_REG <= 8'hec;
143         8'h84: C_REG <= 8'h5f;
144         8'h85: C_REG <= 8'h97;
145         8'h86: C_REG <= 8'h44;
146         8'h87: C_REG <= 8'h17;
147         8'h88: C_REG <= 8'hc4;
148         8'h89: C_REG <= 8'ha7;
149         8'h8a: C_REG <= 8'h7e;
150         8'h8b: C_REG <= 8'h3d;
151         8'h8c: C_REG <= 8'h64;
152         8'h8d: C_REG <= 8'h5d;
153         8'h8e: C_REG <= 8'h19;
154         8'h8f: C_REG <= 8'h73;
155         8'h90: C_REG <= 8'h60;
156         8'h91: C_REG <= 8'h81;
157         8'h92: C_REG <= 8'h4f;
158         8'h93: C_REG <= 8'hdc;
159         8'h94: C_REG <= 8'h22;
160         8'h95: C_REG <= 8'h2a;
161         8'h96: C_REG <= 8'h90;
162         8'h97: C_REG <= 8'h88;
```

```
163         8'h98: C_REG <= 8'h46;
164         8'h99: C_REG <= 8'hee;
165         8'h9a: C_REG <= 8'hb8;
166         8'h9b: C_REG <= 8'h14;
167         8'h9c: C_REG <= 8'hde;
168         8'h9d: C_REG <= 8'h5e;
169         8'h9e: C_REG <= 8'h0b;
170         8'h9f: C_REG <= 8'hdb;
171         8'ha0: C_REG <= 8'he0;
172         8'ha1: C_REG <= 8'h32;
173         8'ha2: C_REG <= 8'h3a;
174         8'ha3: C_REG <= 8'h0a;
175         8'ha4: C_REG <= 8'h49;
176         8'ha5: C_REG <= 8'h06;
177         8'ha6: C_REG <= 8'h24;
178         8'ha7: C_REG <= 8'h5c;
179         8'ha8: C_REG <= 8'hc2;
180         8'ha9: C_REG <= 8'hd3;
181         8'haa: C_REG <= 8'hac;
182         8'hab: C_REG <= 8'h62;
183         8'hac: C_REG <= 8'h91;
184         8'had: C_REG <= 8'h95;
185         8'hae: C_REG <= 8'he4;
186         8'haf: C_REG <= 8'h79;
187         8'hb0: C_REG <= 8'he7;
188         8'hb1: C_REG <= 8'hc8;
189         8'hb2: C_REG <= 8'h37;
190         8'hb3: C_REG <= 8'h6d;
191         8'hb4: C_REG <= 8'h8d;
192         8'hb5: C_REG <= 8'hd5;
193         8'hb6: C_REG <= 8'h4e;
194         8'hb7: C_REG <= 8'ha9;
195         8'hb8: C_REG <= 8'h6c;
196         8'hb9: C_REG <= 8'h56;
197         8'hba: C_REG <= 8'hf4;
198         8'hbb: C_REG <= 8'hea;
199         8'hbc: C_REG <= 8'h65;
200         8'hbd: C_REG <= 8'h7a;
201         8'hbe: C_REG <= 8'hae;
202         8'hbf: C_REG <= 8'h08;
203         8'hc0: C_REG <= 8'hba;
204         8'hc1: C_REG <= 8'h78;
205         8'hc2: C_REG <= 8'h25;
206         8'hc3: C_REG <= 8'h2e;
207         8'hc4: C_REG <= 8'h1c;
208         8'hc5: C_REG <= 8'ha6;
209         8'hc6: C_REG <= 8'hb4;
210         8'hc7: C_REG <= 8'hc6;
211         8'hc8: C_REG <= 8'he8;
212         8'hc9: C_REG <= 8'hdd;
213         8'hca: C_REG <= 8'h74;
214         8'hcb: C_REG <= 8'h1f;
215         8'hcc: C_REG <= 8'h4b;
216         8'hcd: C_REG <= 8'hbd;
217         8'hce: C_REG <= 8'h8b;
218         8'hcf: C_REG <= 8'h8a;
```

```
219         8'hd0: C_REG <= 8'h70;
220         8'hd1: C_REG <= 8'h3e;
221         8'hd2: C_REG <= 8'hb5;
222         8'hd3: C_REG <= 8'h66;
223         8'hd4: C_REG <= 8'h48;
224         8'hd5: C_REG <= 8'h03;
225         8'hd6: C_REG <= 8'hf6;
226         8'hd7: C_REG <= 8'h0e;
227         8'hd8: C_REG <= 8'h61;
228         8'hd9: C_REG <= 8'h35;
229         8'hda: C_REG <= 8'h57;
230         8'hdb: C_REG <= 8'hb9;
231         8'hdc: C_REG <= 8'h86;
232         8'hdd: C_REG <= 8'hc1;
233         8'hde: C_REG <= 8'h1d;
234         8'hdf: C_REG <= 8'h9e;
235         8'he0: C_REG <= 8'he1;
236         8'he1: C_REG <= 8'hf8;
237         8'he2: C_REG <= 8'h98;
238         8'he3: C_REG <= 8'h11;
239         8'he4: C_REG <= 8'h69;
240         8'he5: C_REG <= 8'hd9;
241         8'he6: C_REG <= 8'h8e;
242         8'he7: C_REG <= 8'h94;
243         8'he8: C_REG <= 8'h9b;
244         8'he9: C_REG <= 8'h1e;
245         8'hea: C_REG <= 8'h87;
246         8'heb: C_REG <= 8'he9;
247         8'hec: C_REG <= 8'hce;
248         8'hed: C_REG <= 8'h55;
249         8'hee: C_REG <= 8'h28;
250         8'hef: C_REG <= 8'hdf;
251         8'hf0: C_REG <= 8'h8c;
252         8'hf1: C_REG <= 8'ha1;
253         8'hf2: C_REG <= 8'h89;
254         8'hf3: C_REG <= 8'h0d;
255         8'hf4: C_REG <= 8'hbf;
256         8'hf5: C_REG <= 8'he6;
257         8'hf6: C_REG <= 8'h42;
258         8'hf7: C_REG <= 8'h68;
259         8'hf8: C_REG <= 8'h41;
260         8'hf9: C_REG <= 8'h99;
261         8'hfa: C_REG <= 8'h2d;
262         8'hfb: C_REG <= 8'h0f;
263         8'hfc: C_REG <= 8'hb0;
264         8'hfd: C_REG <= 8'h54;
265         8'hfe: C_REG <= 8'hbb;
266         8'hff: C_REG <= 8'h16;
267       endcase
268     end
269
270     assign C = C_REG;
271
272 endmodule
```

A.1.6 KEY GENERATION

```
1       module GENERATE_KEY(
2       input clk,
3       input [3:0] ROUND_KEY,        // Round Number
4       input [127:0] IN_KEY,    // Input Key
5       output [127:0] OUT_KEY,   // Output Key
6       output reg [127:0] OUT_KEY_R
7   );
8
9       wire [31:0] KEY0, KEY1, KEY2, KEY3;
10      wire [31:0] C;
11
12      assign KEY3 = IN_KEY[127:96];
13      assign KEY2 = IN_KEY[95:64];
14      assign KEY1 = IN_KEY[63:32];
15      assign KEY0 = IN_KEY[31:0];
16
17      assign OUT_KEY[127:96] = KEY3 ^ C ^ RCON(ROUND_KEY);
18      assign OUT_KEY[95:64] = KEY3 ^ C ^ RCON(ROUND_KEY) ^ KEY2;
19      assign OUT_KEY[63:32] = KEY3 ^ C ^ RCON(ROUND_KEY) ^ KEY2 ^ KEY1;
20      assign OUT_KEY[31:0] = KEY3 ^ C ^ RCON(ROUND_KEY) ^ KEY2 ^ KEY1 ^
            KEY0;
21
22      FORWARD_SUBSTITUTION_BOX INST0(.clk(clk), .A(KEY0[23:16]), .C(C
            [31:24]));
23      FORWARD_SUBSTITUTION_BOX INST1(.clk(clk), .A(KEY0[15:8]), .C(C
            [23:16]));
24      FORWARD_SUBSTITUTION_BOX INST2(.clk(clk), .A(KEY0[7:0]), .C(C
            [15:8]));
25      FORWARD_SUBSTITUTION_BOX INST3(.clk(clk), .A(KEY0[31:24]), .C(C
            [7:0]));
26
27      always @ (posedge clk) OUT_KEY_R <= OUT_KEY;
28
29      function [31:0] RCON;
30          input [3:0] ROUND_KEY;
31          case (ROUND_KEY)
32              4'h0: RCON = 32'h01_00_00_00;
33              4'h1: RCON = 32'h02_00_00_00;
34              4'h2: RCON = 32'h04_00_00_00;
35              4'h3: RCON = 32'h08_00_00_00;
36              4'h4: RCON = 32'h10_00_00_00;
37              4'h5: RCON = 32'h20_00_00_00;
38              4'h6: RCON = 32'h40_00_00_00;
39              4'h7: RCON = 32'h80_00_00_00;
40              4'h8: RCON = 32'h1b_00_00_00;
41              4'h9: RCON = 32'h36_00_00_00;
42              default: RCON = 32'h00_00_00_00;
43          endcase
44      endfunction
45  endmodule
```

A.1.7 ROUND-ITERATION

```
1       module ROUND_ITERATION(
```

```
    input clk,
    input [127:0] IN_DATA,
    input [127:0] IN_KEY,
    output reg [127:0] OUT_DATA
);

wire [127:0] SB_DATA, SHIFT_DATA, MIXED_DATA;

SUB_BYTES SB(.clk(clk), .IN_DATA(IN_DATA), .SB_DATA(SB_DATA));
SHIFT_ROWS SR(.clk(clk), .IN_DATA(SB_DATA), .SHIFT_DATA(SHIFT_DATA));
MIX_COLUMNS MC(.clk(clk), .IN_DATA(SHIFT_DATA), .MIXED_DATA(
    MIXED_DATA));

always @(posedge clk) OUT_DATA <= IN_KEY ^ MIXED_DATA;

endmodule
```

A.1.8 LAST ROUND

```
    module LAST_ROUND(
    input clk,
    input [127:0] IN_DATA,
    input [127:0] IN_KEY,
    output reg [127:0] OUT_DATA
);

wire [127:0] SB_DATA, SHIFT_DATA, OUT_KEY;

SUB_BYTES SB(.clk(clk), .IN_DATA(IN_DATA), .SB_DATA(SB_DATA));
SHIFT_ROWS SR(.clk(clk), .IN_DATA(SB_DATA), .SHIFT_DATA(SHIFT_DATA));

always @(posedge clk) OUT_DATA <= IN_KEY ^ SHIFT_DATA;

endmodule
```

A.2 GCM

A.2.1 TOP-MODULE

```
    module AES_GCM_TOP #(parameter length_plain = 384,
            length_tag = 32,
            length_aad = 128)(
input clk,
input reset,
input [length_plain - 1 : 0] Plain_Text,

input [95:0] IV,
input [length_aad -1 :0] AAD,


input [31:0] IN_KEY0,
input [31:0] IN_KEY1,
input [31:0] IN_KEY2,
input [31:0] IN_KEY3,

```

```
17 output [128:0] Cipher,
18 output [length_tag -1 :0] Tag
19
20     );
21
22     wire [127:0] h_key;
23
24     wire [127:0] Initial_CB;
25
26     wire [127:0] Y_0;
27     wire [127:0] Y_1;
28     wire [127:0] Y_2;
29     wire [383:0] Y;
30
31     wire [511:0] g_hash_inp;
32
33     wire [127:0] GH_0;
34     wire [127:0] GH_1;
35     wire [127:0] GH_2;
36
37
38     wire [127:0] Tag_g;
39
40     hash_key h1 (clk, reset, IN_KEY0, IN_KEY1, IN_KEY2, IN_KEY3,
           h_key);
41
42     ICB IC1 ( IV, Initial_CB);
43
44    //integer i = length_plain/128;
45    //integer j;
46    // for ( j = 1; j <= 3 ; j = j+1)
47    // begin
48     GCTR G1( clk, reset, IV,  Plain_Text [127 : 0], Initial_CB,
           IN_KEY0, IN_KEY1, IN_KEY2, IN_KEY3, Y_0);
49     assign Initial_CB = Initial_CB + 1;
50     GCTR G2( clk, reset, IV,  Plain_Text [255 : 128], Initial_CB,
           IN_KEY0, IN_KEY1, IN_KEY2, IN_KEY3, Y_1);
51     assign Initial_CB = Initial_CB + 1;
52     GCTR G3( clk, reset, IV,  Plain_Text [383: 256], Initial_CB,
           IN_KEY0, IN_KEY1, IN_KEY2, IN_KEY3, Y_2);
53
54     assign Y = {Y_0, Y_1, Y_2};
55
56     assign g_hash_inp = {AAD,Y};
57
58     GHASH gh1 (clk, reset, AAD, 128'b0, h_key, IN_KEY0, IN_KEY1,
           IN_KEY2, IN_KEY3, GH_0);
59     GHASH gh2 (clk, reset, Y_0, GH_0, h_key, IN_KEY0, IN_KEY1,
           IN_KEY2, IN_KEY3, GH_1);
60     GHASH gh3 (clk, reset, Y_1, GH_1, h_key, IN_KEY0, IN_KEY1,
           IN_KEY2, IN_KEY3, GH_2);
61     GHASH gh4 (clk, reset, Y_2, GH_2, h_key, IN_KEY0, IN_KEY1,
           IN_KEY2, IN_KEY3, Cipher);
62
63     GCTR T1(clk, reset, IV, Cipher, Initial_CB, IN_KEY0, IN_KEY1,
           IN_KEY2, IN_KEY3, Tag_g);
```

```
64
65      assign Tag = Tag_g [127:96];
66
67  endmodule
```

A.2.2 ICB initialization

```
1       module ICB(input [95:0] IV, output [127:0] Initial_CB);
2   assign Initial_CB = {IV,{31{1'b0}},1'b1};
3   endmodule
```

A.2.3 GCTR

```
1       module GCTR(
2    input clk,
3    input reset,
4    input [95:0] IV,
5    input [127:0] X,
6    input [127:0] Initial_CB,
7
8    input [31:0] IN_KEY0,
9    input [31:0] IN_KEY1,
10   input [31:0] IN_KEY2,
11   input [31:0] IN_KEY3,
12
13   output [127:0] Y
14
15
16
17  );
18
19  wire [31:0] OUT_DATA0;
20  wire [31:0] OUT_DATA1;
21  wire [31:0] OUT_DATA2;
22  wire [31:0] OUT_DATA3;
23  //ICB IC1 ( IV, Initial_CB);
24
25  AES128 Cipher (
26  clk,
27  reset,
28
29  Initial_CB [31:0],
30  Initial_CB [63:32],
31  Initial_CB [95:64],
32  Initial_CB [127:96],
33
34  IN_KEY0,
35  IN_KEY1,
36  IN_KEY2,
37  IN_KEY3,
38
39  OUT_DATA0,
40  OUT_DATA1,
41  OUT_DATA2,
42  OUT_DATA3);
43
```

```
wire [127:0] OUT_DATA;
assign  OUT_DATA = {OUT_DATA3, OUT_DATA2, OUT_DATA1, OUT_DATA0} ;

assign Y = OUT_DATA ^ X ;

endmodule
```

A.2.4 GHASH

```
    module GHASH(
input clk,
input reset,
input [127:0] X,
input [127:0] I,

input [127:0] h_key,

input [31:0] IN_KEY0,
input [31:0] IN_KEY1,
input [31:0] IN_KEY2,
input [31:0] IN_KEY3,

output wire [127:0] Y
);

wire [127:0] mult_in;

assign mult_in = X ^ I;

//hash_key h1 (clk, reset, IN_KEY0, IN_KEY1, IN_KEY2, IN_KEY3, h_key)
    ;

mult_in_GF128 G1( mult_in ,h_key,Y);
endmodule
```

A.2.5 Multiplication in Galois Field

```
      module mult_in_GF128(
        input wire[127:0] X,
        input wire[127:0] Y,
        output reg[127:0] Z
    );
    reg[127:0] V;
    reg[127:0] R;
    integer i;
    always @ (*) begin
        R = {8'b11100001,{120{1'b0}}};

        Z = 0;
        V = X;
        for(i=0;i<128;i=i+1) begin
            if(Y[127-i]==1) begin
                Z = Z^V;
            end
            if(V[0]==0) begin
                V = V>>1;
```

```
20                 end
21                 else begin
22                     V = (V>>1)^R;
23                 end
24             end
25         end
26
27 endmodule
```

A.3 ECDSA

```
1     `define HIGH 1'b1
2      `define LOW 1'b0
3      `define ON 1'b1
4      `define OFF 1'b0
5  package ecdsa_package;
6      //Definitions
7
8
9      //Type Definition
10
11
12     //Parameter
13     parameter ORDER = 256;
14
15     //Basic ALU Operations
16     parameter ADD = 3'h0, SUB = 3'h1, MUL = 3'h2, DIV = 3'h3, MOD =
           3'h4, COM = 3'h5;
17     parameter MOD_INV = 3'h6;
18
19     //Modular ALU Counter
20     parameter CNT_LEN = 4;
21     parameter MOD_ADD_DELAY = 1, MOD_SUB_DELAY = 1, MOD_MUL_DELAY =
           1, MOD_DIV_DELAY = 1, MOD_MOD_DELAY = 1, MOD_COM_DELAY = 1;
22
23     parameter LOAD = 1'b0, COUNT = 1'b1;
24
25     parameter POINT = 1'b0, TANGENT = 1'b1;
26
27     parameter BIT_LEN = 3;
28     parameter BIT_COUNTER_LEN = 3'b101;
29     parameter SLOPE_CALC = 3'b000, POINT_CALC = 3'b001, MATH = 3'b010
           ;
30
31 endpackage
32
33 `timescale 1ns / 1ps
34 //
       //////////////////////////////////////////////////////////////////////////////////
35 // Company:
36 // Engineer:
37 //
38 // Create Date: 04/14/2023 10:55:41 AM
39 // Design Name:
```

```
40 // Module Name: ecdsa_basic_alu
41 // Project Name:
42 // Target Devices:
43 // Tool Versions:
44 // Description:
45 //
46 // Dependencies:
47 //
48 // Revision:
49 // Revision 0.01 - File Created
50 // Additional Comments:
51 //
52 //
    //////////////////////////////////////////////////////////////////////////////////

53
54 import ecdsa_package::*;
55
56 module ecdsa_basic_alu(
57     input enable,
58     input [2:0] op,
59     input [ORDER-1:0] in1,in2,
60     input [ORDER-1:0] p,
61     output reg [2*ORDER-1:0] out
62     );
63
64     always @(*) begin
65         if(enable == `ON) begin
66             case(op)
67                 ADD: out = in1+in2;
68                 SUB: out = in1-in2;
69                 MOD: out = {in1,in2}%p;
70                 MUL: out = in1*in2;
71                 DIV: out = in1/in2;
72                 COM: out[0] = in1>in2;
73             endcase
74         end
75         else begin
76             out = out;
77         end
78     end
79
80 endmodule
81
82 module ecdsa_modular_alu(
83     input clk,
84     input enable,
85     input start,
86     input [1:0] op,
87     input [ORDER-1:0] in1,in2,
88     input [ORDER-1:0] p,
89     output reg [ORDER-1:0] out,
90     output reg done
91     );
92
93
```

```
94      reg basic_alu_enable;
95      reg [ORDER-1:0] basic_alu_in1, basic_alu_in2;
96      reg [2:0] basic_alu_op;
97      wire [2*ORDER-1:0] basic_alu_out_w;
98      reg [2*ORDER-1:0] basic_alu_out;
99
100     ecdsa_basic_alu basic_alu(
101         .enable(basic_alu_enable),
102         .op(basic_alu_op),
103         .in1(basic_alu_in1),
104         .in2(basic_alu_in2),
105         .p(p),
106         .out(basic_alu_out_w)
107     );
108
109     reg [CNT_LEN-1:0] counter;
110     reg counter_state;
111
112     always @(posedge clk) begin
113         if(counter_state == COUNT) begin
114             counter = counter - 1'b1;
115         end
116         else begin
117             case(basic_alu_op)
118                 default: counter = 1;
119                 ADD: counter = MOD_ADD_DELAY;
120                 SUB: counter = MOD_SUB_DELAY;
121                 MUL: counter = MOD_MUL_DELAY;
122                 DIV: counter = MOD_DIV_DELAY;
123                 MOD: counter = MOD_MOD_DELAY;
124                 COM: counter = MOD_COM_DELAY;
125             endcase
126         end
127     end
128
129     reg comp_flag;
130
131     parameter IDLE = 5'h00;
132     parameter MOD_ADD_0 = 5'h01, MOD_ADD_1 = 5'h02, MOD_ADD_2 = 5'h03
        ;
133     parameter MOD_SUB_0 = 5'h04, MOD_SUB_1 = 5'h05, MOD_SUB_2 = 5'h06
        ;
134     parameter MOD_MUL_0 = 5'h07, MOD_MUL_1 = 5'h08, MOD_MUL_2 = 5'h09
        ;
135     parameter MOD_DIV_0 = 5'h0a, MOD_DIV_1 = 5'h0b, MOD_DIV_2 = 5'h0c
        ;
136     parameter MOD_MOD_0 = 5'h0d, MOD_MOD_1 = 5'h0e, MOD_MOD_2 = 5'h0f
        ;
137     parameter COMPARE_0 = 5'h10, COMPARE_1 = 5'h11, COMPARE_2 = 5'h12
        ;
138     parameter ADD_IN  = 5'h13;
139     reg [4:0] state, next_state;
140
141     always @(posedge clk) begin
142         state = next_state;
143     end
```

```
144
145     always @(*) begin
146         if(enable == 'ON) begin
147             if(start == 'HIGH) begin
148                 case(state)
149                     default: next_state = IDLE;
150                     IDLE: begin
151                         case(op)
152                             ADD: next_state = MOD_ADD_0;
153                             SUB: next_state = COMPARE_0;
154                             MUL: next_state = MOD_MUL_0;
155                             DIV: next_state = MOD_DIV_0;
156                             default: next_state = IDLE;
157                         endcase
158                     end
159                     MOD_ADD_0: next_state = MOD_ADD_1;
160                     MOD_ADD_1: begin
161                         if(counter == {CNT_LEN{1'b0}}) next_state =
                                MOD_ADD_2;
162                         else next_state = MOD_ADD_1;
163                     end
164                     MOD_ADD_2: next_state = MOD_MOD_0;
165                     MOD_SUB_0: next_state = MOD_SUB_1;
166                     MOD_SUB_1: begin
167                         if(counter == {CNT_LEN{1'b0}}) next_state =
                                MOD_SUB_2;
168                         else next_state = MOD_SUB_1;
169                     end
170                     MOD_SUB_2: begin
171                         if(comp_flag) next_state = MOD_MOD_0;
172                         else next_state = ADD_IN;
173                     end
174                     MOD_MUL_0: next_state = MOD_MUL_1;
175                     MOD_MUL_1: begin
176                         if(counter == {CNT_LEN{1'b0}}) next_state =
                                MOD_MUL_2;
177                         else next_state = MOD_MUL_1;
178                     end
179                     MOD_MUL_2: next_state = MOD_MOD_0;
180                     MOD_DIV_0: next_state = MOD_DIV_1;
181                     MOD_DIV_1: begin
182                         if(counter == {CNT_LEN{1'b0}}) next_state =
                                MOD_DIV_2;
183                         else next_state = MOD_DIV_1;
184                     end
185                     MOD_DIV_2: next_state = MOD_MOD_0;
186                     MOD_MOD_0: next_state = MOD_MOD_1;
187                     MOD_MOD_1: begin
188                         if(counter == {CNT_LEN{1'b0}}) next_state =
                                MOD_MOD_2;
189                         else next_state = MOD_MOD_1;
190                     end
191                     MOD_MOD_2: next_state = IDLE;
192                     COMPARE_0: next_state = COMPARE_1;
193                     COMPARE_1: begin
194                         if(counter == {CNT_LEN{1'b0}}) next_state =
```

```
                                COMPARE_2;
195                         else next_state = COMPARE_1;
196                     end
197                     COMPARE_2: next_state = MOD_SUB_0;
198                     ADD_IN: next_state = MOD_ADD_1;
199                 endcase
200             end
201             else next_state = IDLE;
202         end
203         else begin
204             next_state = IDLE;
205         end
206     end
207
208     always @(state) begin
209         case(state)
210             IDLE: begin
211                 basic_alu_enable = 'OFF;
212                 done = 'LOW;
213                 out = out;
214             end
215             MOD_ADD_0: begin
216                 basic_alu_enable = 'ON;
217                 basic_alu_in1 = in1;
218                 basic_alu_in2 = in2;
219                 basic_alu_op = {1'b0,op};
220                 counter_state = LOAD;
221             end
222             MOD_ADD_1: counter_state = COUNT;
223             MOD_ADD_2: begin
224                 basic_alu_out = basic_alu_out_w;
225                 basic_alu_enable = 'OFF;
226             end
227             MOD_SUB_0: begin
228                 basic_alu_enable = 'ON;
229                 basic_alu_op = SUB;
230                 counter_state = LOAD;
231             end
232             MOD_SUB_1: counter_state = COUNT;
233             MOD_SUB_2: begin
234                 basic_alu_out = basic_alu_out_w;
235                 basic_alu_enable = 'OFF;
236             end
237             MOD_MUL_0: begin
238                 basic_alu_enable = 'ON;
239                 basic_alu_in1 = in1;
240                 basic_alu_in2 = in2;
241                 basic_alu_op = {1'b0,op};
242                 counter_state = LOAD;
243             end
244             MOD_MUL_1: counter_state = COUNT;
245             MOD_MUL_2: begin
246                 basic_alu_out = basic_alu_out_w;
247                 basic_alu_enable = 'OFF;
248             end
249             MOD_DIV_0: begin
```

```
250                 basic_alu_enable = `ON;
251                 basic_alu_in1 = in1;
252                 basic_alu_in2 = in2;
253                 basic_alu_op = {1'b0,op};
254                 counter_state = LOAD;
255             end
256             MOD_DIV_1: counter_state = COUNT;
257             MOD_DIV_2: begin
258                 basic_alu_out = basic_alu_out_w;
259                 basic_alu_enable = `OFF;
260             end
261             MOD_MOD_0: begin
262                 basic_alu_enable = `ON;
263                 {basic_alu_in1,basic_alu_in2} = basic_alu_out;
264                 basic_alu_op = MOD;
265                 counter_state = LOAD;
266             end
267             MOD_MOD_1: counter_state = COUNT;
268             MOD_MOD_2: begin
269                 basic_alu_out = basic_alu_out_w;
270                 out = basic_alu_out[ORDER-1:0];
271                 basic_alu_enable = `OFF;
272                 done = `HIGH;
273             end
274             COMPARE_0: begin
275                 basic_alu_enable = `ON;
276                 basic_alu_in1 = in1;
277                 basic_alu_in2 = in2;
278                 basic_alu_op = COM;
279                 counter_state = LOAD;
280             end
281             COMPARE_1: begin
282                 counter_state = COUNT;
283             end
284             COMPARE_2: begin
285                 comp_flag = basic_alu_out_w[0];
286                 if(basic_alu_out_w[0]) begin
287                     basic_alu_in1 = in1;
288                     basic_alu_in2 = in2;
289                 end
290                 else begin
291                     basic_alu_in1 = p;
292                     basic_alu_in2 = in2;
293                 end
294                 basic_alu_enable = `OFF;
295             end
296             ADD_IN: begin
297                 comp_flag = `HIGH;
298                 basic_alu_enable = `ON;
299                 basic_alu_in1 = in1;
300                 basic_alu_in2 = basic_alu_out;
301                 basic_alu_op = ADD;
302                 counter_state = LOAD;
303             end
304         endcase
305     end
```

```
306
307 endmodule
308
309 module ecdsa_inverse_modulo(
310     //Inverse Modulo Port
311     input clk,
312     input enable,
313     input start,
314     input [ORDER-1:0] in,
315     input [ORDER-1:0] p,
316     output reg [ORDER-1:0] out,
317     output reg done,
318     //Modular ALU Ports
319     output reg modular_alu_start,
320     output reg [1:0] modular_alu_op,
321     output reg [ORDER-1:0] modular_alu_in1, modular_alu_in2,
322     input [ORDER-1:0] modular_alu_out,
323     input modular_alu_done
324     );
325
326     reg loop_run;
327     parameter R = 1'b0, S = 1'b1;
328
329     parameter IDLE = 4'h0;
330     parameter INITIALIZE = 4'h1;
331     parameter LOOP_DIV_0 = 4'h2, LOOP_DIV_1 = 4'h3, LOOP_DIV_2 = 4'ha
        ;
332     parameter LOOP_MUL_0 = 4'h4, LOOP_MUL_1 = 4'h5, LOOP_MUL_2 = 4'hc
        ;
333     parameter LOOP_SUB_0 = 4'h6, LOOP_SUB_1 = 4'h7, LOOP_SUB_2 = 4'hd
        ;
334     parameter UPDATE = 4'hb;
335     parameter RELOOP = 4'hf, CHECK_EXIT = 4'h8;
336     parameter EXIT = 4'h9;
337
338     reg [ORDER-1:0] q;
339     reg [ORDER-1:0] r_new, r_old, r_temp;
340     reg [ORDER-1:0] s_new, s_old, s_temp;
341     reg [ORDER-1:0] t_new, t_old, t_temp;
342     reg [3:0] state, next_state;
343
344     always @(*) begin
345         if(enable) begin
346             if(start) begin
347                 case(state)
348                     default: next_state = IDLE;
349                     IDLE: next_state = INITIALIZE;
350                     INITIALIZE: next_state = LOOP_DIV_0;
351                     LOOP_DIV_0: next_state = LOOP_DIV_1;
352                     LOOP_DIV_1: begin
353                         if(modular_alu_done) next_state = LOOP_DIV_2;
354                         else next_state = LOOP_DIV_1;
355                     end
356                     LOOP_DIV_2: next_state = LOOP_MUL_0;
357                     LOOP_MUL_0: next_state = LOOP_MUL_1;
358                     LOOP_MUL_1: begin
```

```
359                         if(modular_alu_done) next_state = LOOP_MUL_2;
360                         else next_state = LOOP_MUL_1;
361                     end
362                     LOOP_MUL_2: next_state = LOOP_SUB_0;
363                     LOOP_SUB_0: next_state = LOOP_SUB_1;
364                     LOOP_SUB_1: begin
365                         if(modular_alu_done) next_state = LOOP_SUB_2;
366                         else next_state = LOOP_SUB_1;
367                     end
368                     LOOP_SUB_2: next_state = UPDATE;
369                     UPDATE: begin
370                         next_state = RELOOP;
371                     end
372                     RELOOP: begin
373                         if(loop_run==R) next_state = CHECK_EXIT;
374                         else next_state = LOOP_MUL_0;
375                     end
376                     CHECK_EXIT: begin
377                         if(r_new=={ORDER{1'b0}}) next_state = EXIT;
378                         else next_state = LOOP_DIV_0;
379 
380                     end
381                     EXIT: next_state = IDLE;
382                 endcase
383             end
384             else next_state = IDLE;
385         end
386         else begin
387             next_state = IDLE;
388         end
389     end
390 
391     always @(posedge clk) state <= next_state;
392 
393     always @(state) begin
394         case(state)
395             IDLE: begin
396                 done = `LOW;
397             end
398             INITIALIZE: begin
399                 s_new = {ORDER{1'b0}};
400                 s_old = {{(ORDER-1){1'b0}},1'b1};
401                 r_old = in;
402                 r_new = p;
403             end
404             LOOP_DIV_0: begin
405                 modular_alu_in1 = r_old;
406                 modular_alu_in2 = r_new;
407                 modular_alu_op = DIV;
408                 modular_alu_start = `HIGH;
409             end
410             LOOP_DIV_2: begin
411                 q = modular_alu_out;
412                 modular_alu_start = `LOW;
413                 t_old = r_old;
414                 t_new = r_new;
```

```
415                 loop_run = R;
416             end
417             LOOP_MUL_0: begin
418                 modular_alu_in1 = t_new;
419                 modular_alu_in2 = q;
420                 modular_alu_op = MUL;
421                 modular_alu_start = 'HIGH;
422             end
423             LOOP_MUL_2: begin
424                 t_temp = modular_alu_out;
425                 modular_alu_start = 'LOW;
426             end
427             LOOP_SUB_0: begin
428                 modular_alu_in1 = t_old;
429                 modular_alu_in2 = t_temp;
430                 modular_alu_op = SUB;
431                 modular_alu_start = 'HIGH;
432             end
433             LOOP_SUB_2: begin
434                 case(loop_run)
435                     R: begin
436                         r_temp = modular_alu_out;
437                         r_old = r_new;
438                     end
439                     S: begin
440                         s_temp = modular_alu_out;
441                         s_old = s_new;
442                     end
443                 endcase
444                 modular_alu_start = 'LOW;
445             end
446             UPDATE: begin
447                 case(loop_run)
448                     R: begin
449                         r_new = r_temp;
450                         t_new = s_new;
451                         t_old = s_old;
452                         loop_run = S;
453                     end
454                     S: begin
455                         s_new = s_temp;
456                         loop_run = R;
457                     end
458                 endcase
459             end
460             EXIT: begin
461                 out = s_old;
462                 done = 'HIGH;
463             end
464         endcase
465 
466     end
467 endmodule
468 
469 module ecdsa_alu(
470     input clk,
```

```
471     input enable,
472     input start,
473     input [2:0] op,
474     input [ORDER-1:0] in1, in2 , p,
475     output reg [ORDER-1:0] out,
476     output reg done
477     );
478
479     reg [ORDER-1:0] modular_alu_in1, modular_alu_in2;
480     reg [2:0] op_reg;
481     reg modular_alu_start;
482
483     wire [ORDER-1:0] modular_alu_in1_w, modular_alu_in2_w;
484     wire [2:0] op_w;
485     wire modular_alu_start_w;
486
487     wire modular_alu_done;
488     wire [ORDER-1:0] modular_alu_out;
489
490     ecdsa_modular_alu modular_alu(
491         .clk(clk),
492         .in1(modular_alu_in1_w),
493         .in2(modular_alu_in2_w),
494         .op(op_w[1:0]),
495         .start(modular_alu_start_w),
496         .done(modular_alu_done),
497         .enable(enable),
498         .out(modular_alu_out),
499         .p(p)
500     );
501
502     reg inverse_modulo_enable, inverse_modulo_start;
503     wire inverse_modulo_done;
504     wire [ORDER-1:0] inverse_modulo_out;
505     wire inverse_modulo_modular_alu_start_w;
506     wire [1:0] inverse_modulo_modular_alu_op_w;
507     wire [ORDER-1:0] inverse_modulo_modular_alu_in1_w,
        inverse_modulo_modular_alu_in2_w;
508
509     ecdsa_inverse_modulo inverse_modulo(
510         .clk(clk),
511         .enable(inverse_modulo_enable),
512         .start(inverse_modulo_start),
513         .in(in1),
514         .p(p),
515         .done(inverse_modulo_done),
516         .out(inverse_modulo_out),
517         .modular_alu_start(inverse_modulo_modular_alu_start_w),
518         .modular_alu_in1(inverse_modulo_modular_alu_in1_w),
519         .modular_alu_in2(inverse_modulo_modular_alu_in2_w),
520         .modular_alu_op(inverse_modulo_modular_alu_op_w),
521         .modular_alu_done(modular_alu_done),
522         .modular_alu_out(modular_alu_out)
523     );
524
525     reg [3:0] state, next_state;
```

```
526     reg select_line;
527     parameter MOD_ALU = 1'b0, INV_MOD = 1'b1;
528     parameter IDLE = 4'h0;
529     parameter SET_UP = 4'h1;
530     parameter MODULAR_ALU_LOAD = 4'h2, MODULAR_ALU_PROCESS = 4'h3;
531     parameter INVERSE_MODULO_LOAD = 4'h4, INVERSE_MODULO_PROCESS = 4'
            h5;
532     parameter OUTPUT_VALUE = 4'h6;
533
534     always @(*) begin
535         if(enable==`ON) begin
536             if(start==`HIGH) begin
537                 case(state)
538                     default: next_state = IDLE;
539                     IDLE: next_state = SET_UP;
540                     SET_UP: begin
541                         if(op==MOD_INV) next_state =
                                INVERSE_MODULO_LOAD;
542                         else next_state = MODULAR_ALU_LOAD;
543                     end
544                     MODULAR_ALU_LOAD: next_state =
                            MODULAR_ALU_PROCESS;
545                     MODULAR_ALU_PROCESS: begin
546                         if(modular_alu_done) next_state =
                                OUTPUT_VALUE;
547                         else next_state = MODULAR_ALU_PROCESS;
548                     end
549                     INVERSE_MODULO_LOAD: next_state =
                            INVERSE_MODULO_PROCESS;
550                     INVERSE_MODULO_PROCESS: begin
551                         if(inverse_modulo_done) next_state =
                                OUTPUT_VALUE;
552                         else next_state = INVERSE_MODULO_PROCESS;
553                     end
554                     OUTPUT_VALUE: next_state = IDLE;
555                 endcase
556             end
557         end
558         else begin
559             next_state = IDLE;
560         end
561     end
562
563     always@(posedge clk) begin
564         state <= next_state;
565     end
566
567     always @(state) begin
568         case(state)
569             IDLE: begin
570                 done = `LOW;
571                 inverse_modulo_enable = `OFF;
572             end
573             MODULAR_ALU_LOAD: begin
574                 select_line = MOD_ALU;
575                 modular_alu_in1 = in1;
```

```
576                 modular_alu_in2 = in2;
577                 op_reg = op;
578                 modular_alu_start = 'HIGH;
579             end
580             INVERSE_MODULO_LOAD: begin
581                 select_line = INV_MOD;
582                 inverse_modulo_enable = 'ON;
583                 inverse_modulo_start = 'HIGH;
584             end
585             OUTPUT_VALUE: begin
586                 if(select_line == MOD_ALU) begin
587                     modular_alu_start = 'LOW;
588                     out = modular_alu_out;
589                 end
590                 else begin
591                     inverse_modulo_start = 'LOW;
592                     out = inverse_modulo_out;
593                 end
594                 done = 'HIGH;
595             end
596         endcase
597     end
598
599     assign modular_alu_start_w = (select_line==MOD_ALU)?
        modular_alu_start : inverse_modulo_modular_alu_start_w;
600     assign modular_alu_in1_w = (select_line==MOD_ALU)?
        modular_alu_in1 : inverse_modulo_modular_alu_in1_w;
601     assign modular_alu_in2_w = (select_line==MOD_ALU)?
        modular_alu_in2 : inverse_modulo_modular_alu_in2_w;
602     assign op_w = (select_line==MOD_ALU)? op_reg :
        inverse_modulo_modular_alu_op_w;
603
604 endmodule
605
606 module ecdsa_point_multiplication(
607     input clk,
608     input enable,
609     input start,
610     input [ORDER-1:0] scalar_k,
611     input [ORDER-1:0] x, y,
612     output reg [ORDER-1:0] r,
613     output reg done,
614     //Point Calculation
615     output reg point_calc_enable,
616     output reg point_calc_start,
617     output reg point_type,
618     output reg [ORDER-1:0] point_calc_x_p, point_calc_y_p,
619     output reg [ORDER-1:0] point_calc_x_q, point_calc_y_q,
620     input [ORDER-1:0] point_calc_x_r, point_calc_y_r,
621     input  point_calc_done
622     );
623     reg [ORDER-1:0] s;
624     reg [BIT_LEN-1:0] bit_counter;
625
626     reg [ORDER-1:0] x_reg, y_reg;
627     parameter IDLE = 4'h0;
```

```
628     parameter INITIALIZE = 4'h1;
629     //Loop
630     parameter LOOP_POINT_DOUBLE_0 = 4'h2, LOOP_POINT_DOUBLE_1 = 4'h3,
            LOOP_POINT_DOUBLE_2 = 4'h4;
631     parameter LOOP_CHECK_INDEX = 4'h5;
632     parameter LOOP_POINT_ADD_0 = 4'h6, LOOP_POINT_ADD_1 = 4'h7,
            LOOP_POINT_ADD_2 = 4'h8;
633     parameter COUNTER_UPDATE = 4'h9;
634     parameter CHECK_EXIT = 4'ha;
635     parameter EXIT = 4'hb;
636
637     reg [3:0] state, next_state;
638
639     always @(*) begin
640         if(enable) begin
641             if(start) begin
642                 case(state)
643                     default: next_state = IDLE;
644                     IDLE: next_state = INITIALIZE;
645                     INITIALIZE: next_state = LOOP_CHECK_INDEX;
646                     LOOP_CHECK_INDEX: begin
647                         if(scalar_k[bit_counter]) next_state =
                                LOOP_POINT_ADD_0;
648                         else next_state = LOOP_POINT_DOUBLE_0;
649                     end
650                     LOOP_POINT_ADD_0: next_state = LOOP_POINT_ADD_1;
651                     LOOP_POINT_ADD_1: begin
652                         if(point_calc_done == 'HIGH) next_state =
                                LOOP_POINT_ADD_2;
653                         else next_state = LOOP_POINT_ADD_1;
654                     end
655                     LOOP_POINT_ADD_2: next_state =
                            LOOP_POINT_DOUBLE_0;
656                     LOOP_POINT_DOUBLE_0: next_state =
                            LOOP_POINT_DOUBLE_1;
657                     LOOP_POINT_DOUBLE_1: begin
658                         if(point_calc_done == 'HIGH) next_state =
                                LOOP_POINT_DOUBLE_2;
659                         else next_state = LOOP_POINT_DOUBLE_1;
660                     end
661                     LOOP_POINT_DOUBLE_2: next_state = COUNTER_UPDATE;
662                     COUNTER_UPDATE: next_state = CHECK_EXIT;
663                     CHECK_EXIT: begin
664                         if(bit_counter == BIT_COUNTER_LEN) next_state
                                = EXIT;
665                         else next_state = LOOP_CHECK_INDEX;
666                     end
667                 endcase
668             end
669             else begin
670                 next_state = IDLE;
671             end
672         end
673         else begin
674             next_state = IDLE;
675         end
```

```
676     end
677
678     always @(posedge clk) state <= next_state;
679
680     always @(state) begin
681         case(state)
682             IDLE: begin
683                 point_calc_enable = `OFF;
684                 done = `LOW;
685             end
686             INITIALIZE: begin
687                 point_calc_enable = `ON;
688                 bit_counter = {BIT_LEN{1'b0}};
689                 r = {ORDER{1'b0}};
690                 s = {ORDER{1'b0}};
691                 x_reg = x;
692                 y_reg = y;
693             end
694             LOOP_POINT_ADD_0: begin
695                 point_calc_x_p = r;
696                 point_calc_y_p = s;
697                 point_calc_x_q = x_reg;
698                 point_calc_y_q = y_reg;
699                 point_type = POINT;
700                 point_calc_start = `ON;
701             end
702             LOOP_POINT_ADD_2: begin
703                 r = point_calc_x_r;
704                 s = point_calc_y_r;
705                 point_calc_start = `OFF;
706             end
707             LOOP_POINT_DOUBLE_0: begin
708                 point_calc_x_p = x_reg;
709                 point_calc_y_p = y_reg;
710                 point_type = TANGENT;
711                 point_calc_start = `HIGH;
712             end
713             LOOP_POINT_DOUBLE_2: begin
714                 x_reg = point_calc_x_r;
715                 y_reg = point_calc_y_r;
716                 point_calc_start = `OFF;
717             end
718             COUNTER_UPDATE: begin
719                 bit_counter = bit_counter+1'b1;
720             end
721             EXIT: begin
722                 done = `HIGH;
723             end
724         endcase
725     end
726
727 endmodule
728
729 module point_calc(
730     input clk,
731     input enable,
```

```
732     input start,
733     input [ORDER-1:0] x_p, y_p,
734     input [ORDER-1:0] x_q, y_q,
735     input point_type,
736     output reg [ORDER-1:0] x_r, y_r,
737     output reg done,
738     //Slope Calc
739     output reg slope_calc_enable,
740     output reg slope_calc_start,
741     output [ORDER-1:0] slope_x_p, slope_y_p,
742     output [ORDER-1:0] slope_x_q, slope_y_q,
743     output slope_type,
744     input [ORDER-1:0] m,
745     input slope_calc_done,
746     //ALU Port
747     output reg alu_enable,
748     output reg [ORDER-1:0] alu_in1,alu_in2,
749     output reg [2:0] op,
750     output reg alu_start,
751     input alu_done,
752     input [ORDER-1:0] alu_out
753     );
754
755     reg [ORDER-1:0] m_reg;
756
757     assign slope_type = point_type;
758     assign {slope_x_p,slope_y_p} = {x_p,y_p};
759     assign {slope_x_q,slope_y_q} = {x_q,y_q};
760
761     parameter IDLE = 5'h00;
762     parameter SLOPE_CALC_0 = 5'h01, SLOPE_CALC_1 = 5'h02,
            SLOPE_CALC_2 = 5'h03;
763     parameter X_R_0 = 5'h04, X_R_1 = 5'h05, X_R_2 = 5'h06;
764     parameter X_R_3 = 5'h07, X_R_4 = 5'h08, X_R_5 = 5'h09;
765     parameter X_R_6 = 5'h0a, X_R_7 = 5'h0b, X_R_8 = 5'h0c;
766     parameter Y_R_0 = 5'h0d, Y_R_1 = 5'h0e, Y_R_2 = 5'h0f;
767     parameter Y_R_3 = 5'h10, Y_R_4 = 5'h11, Y_R_5 = 5'h12;
768     parameter Y_R_6 = 5'h13, Y_R_7 = 5'h14, Y_R_8 = 5'h15;
769     parameter EXIT = 5'h16;
770     parameter ZERO = 5'h17;
771
772     reg [4:0] state, next_state;
773
774     always @(*) begin
775         if(enable==`ON) begin
776             if(start==`HIGH) begin
777                 case(state)
778                     default: next_state = IDLE;
779                     IDLE: begin
780                         if((x_p|y_p)&&(x_q|y_q)) next_state =
                                SLOPE_CALC_0;
781                         else next_state = ZERO;
782                     end
783                     SLOPE_CALC_0: next_state = SLOPE_CALC_1;
784                     SLOPE_CALC_1: begin
785                         if(slope_calc_done) next_state = SLOPE_CALC_2
```

```
                        ;
786                     else next_state = SLOPE_CALC_1;
787                 end
788                 SLOPE_CALC_2: next_state = X_R_0;
789                 X_R_0: next_state = X_R_1;
790                 X_R_1: begin
791                     if(alu_done) next_state = X_R_2;
792                     else next_state = X_R_1;
793                 end
794                 X_R_2: next_state = X_R_3;
795                 X_R_3: next_state = X_R_4;
796                 X_R_4: begin
797                     if(alu_done) next_state = X_R_5;
798                     else next_state = X_R_4;
799                 end
800                 X_R_5: next_state = X_R_6;
801                 X_R_6: next_state = X_R_7;
802                 X_R_7: begin
803                     if(alu_done) next_state = X_R_8;
804                     else next_state = X_R_7;
805                 end
806                 X_R_8: next_state = Y_R_0;
807                 Y_R_0: next_state = Y_R_1;
808                 Y_R_1: begin
809                     if(alu_done) next_state = Y_R_2;
810                     else next_state = Y_R_1;
811                 end
812                 Y_R_2: next_state = Y_R_3;
813                 Y_R_3: next_state = Y_R_4;
814                 Y_R_4: begin
815                     if(alu_done) next_state = Y_R_5;
816                     else next_state = Y_R_4;
817                 end
818                 Y_R_5: next_state = Y_R_6;
819                 Y_R_6: next_state = Y_R_7;
820                 Y_R_7: begin
821                     if(alu_done) next_state = Y_R_8;
822                     else next_state = Y_R_7;
823                 end
824                 Y_R_8: next_state = EXIT;
825                 ZERO: next_state = EXIT;
826                 EXIT: next_state = IDLE;
827             endcase
828         end
829         else begin
830             next_state = IDLE;
831         end
832     end
833     else begin
834         next_state = IDLE;
835     end
836 end
837
838 always @(posedge clk) state <= next_state;
839
840 always@(state) begin
```

```
841         case(state)
842             IDLE: begin
843                 done = `LOW;
844                 alu_enable = `OFF;
845             end
846             SLOPE_CALC_0: begin
847                 slope_calc_enable = `ON;
848                 slope_calc_start = `HIGH;
849             end
850             SLOPE_CALC_2: begin
851                 m_reg = m;
852                 slope_calc_start = `LOW;
853                 slope_calc_enable = `OFF;
854             end
855             X_R_0: begin
856                 alu_in1 = m;
857                 alu_in2 = m;
858                 op = MUL;
859                 alu_enable = `ON;
860                 alu_start = `HIGH;
861             end
862             X_R_2: begin
863                 x_r = alu_out;
864                 alu_start = `LOW;
865             end
866             X_R_3: begin
867                 alu_in1 = x_r;
868                 alu_in2 = x_p;
869                 op = SUB;
870                 alu_start = `HIGH;
871             end
872             X_R_5: begin
873                 x_r = alu_out;
874                 alu_start = `LOW;
875             end
876             X_R_6: begin
877                 alu_in1 = x_r;
878                 if(point_type == TANGENT) alu_in2 = x_p;
879                 else alu_in2 = x_q;
880                 op = SUB;
881                 alu_start = `HIGH;
882             end
883             X_R_8: begin
884                 x_r = alu_out;
885                 alu_start = `LOW;
886             end
887             Y_R_0: begin
888                 alu_in1 = x_p;
889                 alu_in2 = x_r;
890                 op = SUB;
891                 alu_start = `HIGH;
892             end
893             Y_R_2: begin
894                 y_r = alu_out;
895                 alu_start = `LOW;
896             end
```

```
897                 Y_R_3: begin
898                     alu_in1 = m_reg;
899                     alu_in2 = y_r;
900                     op = MUL;
901                     alu_start = 'HIGH;
902                 end
903                 Y_R_5: begin
904                     y_r = alu_out;
905                     alu_start = 'LOW;
906                 end
907                 Y_R_6: begin
908                     alu_in1 = y_r;
909                     alu_in2 = y_p;
910                     op = SUB;
911                     alu_start = 'HIGH;
912                 end
913                 Y_R_8: begin
914                     y_r = alu_out;
915                     alu_start = 'LOW;
916                 end
917                 ZERO: begin
918                     x_r = x_q|x_p;
919                     y_r = y_q|y_p;
920                 end
921                 EXIT: begin
922                     done = 'HIGH;
923                 end
924             endcase
925         end
926
927 endmodule
928
929 module slope_calc(
930         input clk,
931         input enable,
932         input start,
933         input [ORDER-1:0] x_p, y_p,
934         input [ORDER-1:0] x_q, y_q,
935         input slope_type,
936         output reg [ORDER-1:0] m,
937         output reg done,
938         input [ORDER-1:0] curve_a,
939         //ALU Port
940         output reg alu_enable,
941         output reg [ORDER-1:0] alu_in1,alu_in2,
942         output reg [2:0] op,
943         output reg alu_start,
944         input alu_done,
945         input [ORDER-1:0] alu_out
946         );
947
948         parameter IDLE = 6'h00;
949         parameter TYPE_BRANCH = 6'h01;
950
951         parameter POINT_SLOPE_0 = 6'h2, POINT_SLOPE_1 = 6'h3,
                POINT_SLOPE_2 = 6'h4;
```

```
952     parameter POINT_SLOPE_3 = 6'h5, POINT_SLOPE_4 = 6'h6,
            POINT_SLOPE_5 = 6'h7;
953     parameter POINT_SLOPE_6 = 6'h8, POINT_SLOPE_7 = 6'h9,
            POINT_SLOPE_8 = 6'ha;
954     parameter POINT_SLOPE_9 = 6'hb, POINT_SLOPE_10 = 6'hc,
            POINT_SLOPE_11 = 6'hd;
955
956     parameter TANGENT_SLOPE_0 = 6'he, TANGENT_SLOPE_1 = 6'hf,
            TANGENT_SLOPE_2 = 6'h10;
957     parameter TANGENT_SLOPE_3 = 6'h11, TANGENT_SLOPE_4 = 6'h12,
            TANGENT_SLOPE_5 = 6'h13;
958     parameter TANGENT_SLOPE_6 = 6'h14, TANGENT_SLOPE_7 = 6'h15,
            TANGENT_SLOPE_8 = 6'h16;
959     parameter TANGENT_SLOPE_9 = 6'h17, TANGENT_SLOPE_10 = 6'h18,
            TANGENT_SLOPE_11 = 6'h19;
960     parameter TANGENT_SLOPE_12 = 6'h1a, TANGENT_SLOPE_13 = 6'h1b,
            TANGENT_SLOPE_14 = 6'h1c;
961     parameter TANGENT_SLOPE_15 = 6'h1d, TANGENT_SLOPE_16 = 6'h1e,
            TANGENT_SLOPE_17 = 6'h1f;
962
963     parameter OUTPUT_VALUE = 6'h2f;
964
965     reg [5:0] state, next_state;
966
967     always @(*) begin
968         if(enable==`ON) begin
969             if(start==`HIGH) begin
970                 case(state)
971                     default: next_state = IDLE;
972                     IDLE: next_state = TYPE_BRANCH;
973                     TYPE_BRANCH: begin
974                         case(slope_type)
975                             POINT: next_state = POINT_SLOPE_0;
976                             TANGENT: next_state = TANGENT_SLOPE_0;
977                         endcase
978                     end
979                     POINT_SLOPE_0: next_state = POINT_SLOPE_1;
980                     POINT_SLOPE_1: begin
981                         if(alu_done) next_state = POINT_SLOPE_2;
982                         else next_state = POINT_SLOPE_1;
983                     end
984                     POINT_SLOPE_2: next_state = POINT_SLOPE_3;
985                     POINT_SLOPE_3: next_state = POINT_SLOPE_4;
986                     POINT_SLOPE_4: begin
987                         if(alu_done) next_state = POINT_SLOPE_5;
988                         else next_state = POINT_SLOPE_4;
989                     end
990                     POINT_SLOPE_5: next_state = POINT_SLOPE_6;
991                     POINT_SLOPE_6: next_state = POINT_SLOPE_7;
992                     POINT_SLOPE_7: begin
993                         if(alu_done) next_state = POINT_SLOPE_8;
994                         else next_state = POINT_SLOPE_7;
995                     end
996                     POINT_SLOPE_8: next_state = POINT_SLOPE_9;
997                     POINT_SLOPE_9: next_state = POINT_SLOPE_10;
998                     POINT_SLOPE_10: begin
```

```
999                         if(alu_done) next_state = POINT_SLOPE_11;
1000                        else next_state = POINT_SLOPE_10;
1001                    end
1002                    POINT_SLOPE_11: next_state = OUTPUT_VALUE;
1003                    TANGENT_SLOPE_0: next_state = TANGENT_SLOPE_1;
1004                    TANGENT_SLOPE_1: begin
1005                        if(alu_done) next_state = TANGENT_SLOPE_2;
1006                        else next_state = TANGENT_SLOPE_1;
1007                    end
1008                    TANGENT_SLOPE_2: next_state = TANGENT_SLOPE_3;
1009                    TANGENT_SLOPE_3: next_state = TANGENT_SLOPE_4;
1010                    TANGENT_SLOPE_4: begin
1011                        if(alu_done) next_state = TANGENT_SLOPE_5;
1012                        else next_state = TANGENT_SLOPE_4;
1013                    end
1014                    TANGENT_SLOPE_5: next_state = TANGENT_SLOPE_6;
1015                    TANGENT_SLOPE_6: next_state = TANGENT_SLOPE_7;
1016                    TANGENT_SLOPE_7: begin
1017                        if(alu_done) next_state = TANGENT_SLOPE_8;
1018                        else next_state = TANGENT_SLOPE_7;
1019                    end
1020                    TANGENT_SLOPE_8: next_state = TANGENT_SLOPE_9;
1021                    TANGENT_SLOPE_9: next_state = TANGENT_SLOPE_10;
1022                    TANGENT_SLOPE_10: begin
1023                        if(alu_done) next_state = TANGENT_SLOPE_11;
1024                        else next_state = TANGENT_SLOPE_10;
1025                    end
1026                    TANGENT_SLOPE_11: next_state = TANGENT_SLOPE_12;
1027                    TANGENT_SLOPE_12: next_state = TANGENT_SLOPE_13;
1028                    TANGENT_SLOPE_13: begin
1029                        if(alu_done) next_state = TANGENT_SLOPE_14;
1030                        else next_state = TANGENT_SLOPE_13;
1031                    end
1032                    TANGENT_SLOPE_14: next_state = TANGENT_SLOPE_15;
1033                    TANGENT_SLOPE_15: next_state = TANGENT_SLOPE_16;
1034                    TANGENT_SLOPE_16: begin
1035                        if(alu_done) next_state = TANGENT_SLOPE_17;
1036                        else next_state = TANGENT_SLOPE_16;
1037                    end
1038                    TANGENT_SLOPE_17: next_state = OUTPUT_VALUE;
1039                endcase
1040            end
1041            else begin
1042                next_state = IDLE;
1043            end
1044        end
1045        else begin
1046            next_state = IDLE;
1047        end
1048    end
1049
1050    always @(posedge clk) state = next_state;
1051
1052    always @(state) begin
1053        case(state)
1054            IDLE: begin
```

```
1055                    done = `LOW;
1056                    alu_enable = `OFF;
1057                end
1058                TYPE_BRANCH: alu_enable = `ON;
1059                POINT_SLOPE_0: begin
1060                    alu_in1 = y_q;
1061                    alu_in2 = y_p;
1062                    op = SUB;
1063                    alu_start = `HIGH;
1064                end
1065                POINT_SLOPE_2: begin
1066                    m = alu_out;
1067                    alu_start = `LOW;
1068                end
1069                POINT_SLOPE_3: begin
1070                    alu_in1 = x_q;
1071                    alu_in2 = x_p;
1072                    op = SUB;
1073                    alu_start = `HIGH;
1074                end
1075                POINT_SLOPE_5: begin
1076                    alu_in1 = alu_out;
1077                    alu_start = `LOW;
1078                end
1079                POINT_SLOPE_6: begin
1080                    op = MOD_INV;
1081                    alu_start = `HIGH;
1082                end
1083                POINT_SLOPE_8: begin
1084                    alu_in2 = alu_out;
1085                    alu_start = `LOW;
1086                end
1087                POINT_SLOPE_9: begin
1088                    alu_in1 = m;
1089                    op = MUL;
1090                    alu_start = `HIGH;
1091                end
1092                POINT_SLOPE_11: begin
1093                    m = alu_out;
1094                    alu_start = `LOW;
1095                end
1096                TANGENT_SLOPE_0: begin
1097                    alu_in1 = x_p;
1098                    alu_in2 = x_p;
1099                    op = MUL;
1100                    alu_start = `HIGH;
1101                end
1102                TANGENT_SLOPE_2: begin
1103                    m = alu_out;
1104                    alu_start = `LOW;
1105                end
1106                TANGENT_SLOPE_3: begin
1107                    alu_in1 = m;
1108                    alu_in2 = {{(ORDER-2){1'b0}},2'b11};
1109                    op = MUL;
1110                    alu_start = `HIGH;
```

```
1111                end
1112                TANGENT_SLOPE_5: begin
1113                    m = alu_out;
1114                    alu_start = `LOW;
1115                end
1116                TANGENT_SLOPE_6: begin
1117                    alu_in1 = m;
1118                    alu_in2 = curve_a;
1119                    op = ADD;
1120                    alu_start = `HIGH;
1121                end
1122                TANGENT_SLOPE_8: begin
1123                    m = alu_out;
1124                    alu_start = `LOW;
1125                end
1126                TANGENT_SLOPE_9: begin
1127                    alu_in1 = y_p;
1128                    alu_in2 = {{(ORDER-2){1'b0}},2'b10};
1129                    op = MUL;
1130                    alu_start = `HIGH;
1131                end
1132                TANGENT_SLOPE_11: begin
1133                    alu_in1 = alu_out;
1134                    alu_start = `LOW;
1135                end
1136                TANGENT_SLOPE_12: begin
1137                    op = MOD_INV;
1138                    alu_start = `HIGH;
1139                end
1140                TANGENT_SLOPE_14: begin
1141                    alu_in2 = alu_out;
1142                    alu_start = `LOW;
1143                end
1144                TANGENT_SLOPE_15: begin
1145                    alu_in1 = m;
1146                    op = MUL;
1147                    alu_start = `HIGH;
1148                end
1149                TANGENT_SLOPE_17: begin
1150                    m = alu_out;
1151                    alu_start = `LOW;
1152                end
1153                OUTPUT_VALUE: done = `HIGH;
1154            endcase
1155        end
1156    endmodule
1157
1158    module ecdsa_main(
1159        input clk,
1160        input enable,
1161        input start,
1162        output reg done,
1163        input [ORDER-1:0] G_x, G_y,
1164        input [ORDER-1:0] p, n, z, a, k,
1165        input [ORDER-1:0] curve_a,
1166        output reg [ORDER-1:0] r,
```

```
1167        output reg [ORDER-1:0] s
1168        );
1169        //ALU
1170        wire alu_enable_w;
1171        wire [ORDER-1:0] in1_w, in2_w;
1172        wire [2:0] op_w;
1173        wire alu_start_w;
1174        wire alu_done_w;
1175        wire [ORDER-1:0] alu_out_w;
1176        wire [ORDER-1:0] mod_p_w;
1177        reg mod_p_select;
1178
1179        ecdsa_alu ecdsa_alu(
1180            .clk(clk),
1181            .enable(alu_enable_w),
1182            .start(alu_start_w),
1183            .in1(in1_w),
1184            .in2(in2_w),
1185            .p(mod_p_w),
1186            .op(op_w),
1187            .out(alu_out_w),
1188            .done(alu_done_w)
1189        );
1190
1191        //Slope to ALU Connections
1192        wire [ORDER-1:0] slope_in1, slope_in2;
1193        wire [2:0] slope_op;
1194        wire slope_start, slope_enable;
1195
1196        //Point to ALU Connections
1197        wire [ORDER-1:0] point_in1, point_in2;
1198        wire [2:0] point_op;
1199        wire point_start, point_enable;
1200
1201        //Math ALU
1202        reg [ORDER-1:0] alu_in1, alu_in2;
1203        reg [2:0] op;
1204        reg alu_start;
1205        reg alu_enable;
1206
1207        //Slope_Calc to Point_Calc Connections
1208        wire slope_calc_enable;
1209        wire slope_calc_start;
1210        wire [ORDER-1:0] slope_x_p, slope_y_p;
1211        wire [ORDER-1:0] slope_x_q, slope_y_q;
1212        wire slope_type;
1213        wire [ORDER-1:0] m;
1214        wire slope_calc_done;
1215
1216        //Point_Calc to Point Multiplication Connections
1217        wire point_calc_enable;
1218        wire point_calc_start;
1219        wire point_type;
1220        wire [ORDER-1:0] point_calc_x_p, point_calc_y_p;
1221        wire [ORDER-1:0] point_calc_x_q, point_calc_y_q;
1222        wire [ORDER-1:0] point_calc_x_r, point_calc_y_r;
```

```
1223        wire point_calc_done;
1224
1225        //Point Multiplication
1226        reg point_multiplication_enable;
1227        reg point_multiplication_start;
1228        wire point_multiplication_done;
1229
1230        ecdsa_point_multiplication point_multiplication(
1231            .clk(clk),
1232            .enable(point_multiplication_enable),
1233            .start(point_multiplication_start),
1234            .done(point_multiplication_done),
1235            .scalar_k(k),
1236            .x(G_x), .y(G_y),
1237            .r(r),
1238            //Point_calc to Point Multiplication
1239            .point_calc_enable(point_calc_enable),
1240            .point_calc_start(point_calc_start),
1241            .point_type(point_type),
1242            .point_calc_x_p(point_calc_x_p), .point_calc_y_p(
                    point_calc_y_p),
1243            .point_calc_x_q(point_calc_x_q), .point_calc_y_q(
                    point_calc_y_q),
1244            .point_calc_x_r(point_calc_x_r), .point_calc_y_r(
                    point_calc_y_r),
1245            .point_calc_done(point_calc_done)
1246        );
1247
1248        point_calc point_calculator(
1249            .clk(clk),
1250            .enable(point_calc_enable),
1251            .start(point_calc_start),
1252            .x_p(point_calc_x_p), .y_p(point_calc_y_p),
1253            .x_q(point_calc_x_q), .y_q(point_calc_y_q),
1254            .x_r(point_calc_x_r), .y_r(point_calc_y_r),
1255            .point_type(point_type),
1256            .done(point_calc_done),
1257            //Point Calc  to Slope Calc
1258            .slope_calc_enable(slope_calc_enable),
1259            .slope_calc_start(slope_calc_start),
1260            .slope_x_p(slope_x_p), .slope_y_p(slope_y_p),
1261            .slope_x_q(slope_x_q), .slope_y_q(slope_y_q),
1262            .slope_type(slope_type),
1263            .m(m),
1264            .slope_calc_done(slope_calc_done),
1265            //Point_calc to ALU
1266            .alu_in1(point_in1), .alu_in2(point_in2),
1267            .op(point_op),
1268            .alu_enable(point_enable),
1269            .alu_start(point_start),
1270            .alu_done(alu_done_w),
1271            .alu_out(alu_out_w)
1272        );
1273
1274        slope_calc slope_calculator(
1275            .clk(clk),
```

```
1276            .enable(slope_calc_enable),
1277            .start(slope_calc_start),
1278            .x_p(slope_x_p), .y_p(slope_y_p),
1279            .x_q(slope_x_q), .y_q(slope_y_q),
1280            .slope_type(slope_type),
1281            .m(m),
1282            .done(slope_calc_done),
1283            .curve_a(curve_a),
1284            //Slope_calc to ALU
1285            .alu_enable(slope_enable),
1286            .alu_start(slope_start),
1287            .alu_in1(slope_in1), .alu_in2(slope_in2),
1288            .op(slope_op),
1289            .alu_done(alu_done_w),
1290            .alu_out(alu_out_w)
1291        );
1292
1293        reg [4:0] state, next_state;
1294
1295        parameter IDLE = 5'h00;
1296        parameter POINT_MULTIPLICATION_0 = 5'h01, POINT_MULTIPLICATION_1
                = 5'h02, POINT_MULTIPLICATION_2 = 5'h03;
1297        parameter R_MOD_0 = 5'h11, R_MOD_1 = 5'h12, R_MOD_2 = 5'h13;
1298        parameter S_MUL_0 = 5'h04, S_MUL_1 = 5'h05, S_MUL_2 = 5'h06;
1299        parameter S_ADD_0 = 5'h07, S_ADD_1 = 5'h08, S_ADD_2 = 5'h09;
1300        parameter S_INV_0 = 5'h0a, S_INV_1 = 5'h0b, S_INV_2 = 5'h0c;
1301        parameter S_PROD_0 = 5'h0d, S_PROD_1 = 5'h0e, S_PROD_2 = 5'h0f;
1302        parameter EXIT = 5'h10;
1303
1304        always @(*) begin
1305            if(enable == `ON) begin
1306                if(start == `HIGH) begin
1307                    case(state)
1308                        default: next_state = IDLE;
1309                        IDLE: next_state = POINT_MULTIPLICATION_0;
1310                        POINT_MULTIPLICATION_0: next_state =
                                POINT_MULTIPLICATION_1;
1311                        POINT_MULTIPLICATION_1: begin
1312                            if(point_multiplication_done) next_state =
                                    POINT_MULTIPLICATION_2;
1313                            else next_state = POINT_MULTIPLICATION_1;
1314                        end
1315                        POINT_MULTIPLICATION_2: next_state = R_MOD_0;
1316                        R_MOD_0: next_state = R_MOD_1;
1317                        R_MOD_1: begin
1318                            if(alu_done_w) next_state = R_MOD_2;
1319                            else next_state = R_MOD_1;
1320                        end
1321                        R_MOD_2: next_state = S_MUL_0;
1322                        S_MUL_0: next_state = S_MUL_1;
1323                        S_MUL_1: begin
1324                            if(alu_done_w) next_state = S_MUL_2;
1325                            else next_state = S_MUL_1;
1326                        end
1327                        S_MUL_2: next_state = S_ADD_0;
1328                        S_ADD_0: next_state = S_ADD_1;
```

```
1329                    S_ADD_1: begin
1330                        if(alu_done_w) next_state = S_ADD_2;
1331                        else next_state = S_ADD_1;
1332                    end
1333                    S_ADD_2: next_state = S_INV_0;
1334                    S_INV_0: next_state = S_INV_1;
1335                    S_INV_1: begin
1336                        if(alu_done_w) next_state = S_INV_2;
1337                        else next_state = S_INV_1;
1338                    end
1339                    S_INV_2: next_state = S_PROD_0;
1340                    S_PROD_0: next_state = S_PROD_1;
1341                    S_PROD_1: begin
1342                        if(alu_done_w) next_state = S_PROD_2;
1343                        else next_state = S_PROD_1;
1344                    end
1345                    S_PROD_2: next_state = EXIT;
1346                    EXIT: next_state = IDLE;
1347                endcase
1348            end
1349        end
1350        else begin
1351            next_state = IDLE;
1352        end
1353    end
1354
1355    always @(posedge clk) state <= next_state;
1356
1357    always @(state) begin
1358        case(state)
1359            IDLE: begin
1360                done = `LOW;
1361                alu_enable = `LOW;
1362                mod_p_select = 1'b1;
1363            end
1364            POINT_MULTIPLICATION_0: begin
1365                point_multiplication_enable = `ON;
1366                point_multiplication_start = `HIGH;
1367            end
1368            POINT_MULTIPLICATION_2: begin
1369                point_multiplication_start = `LOW;
1370                point_multiplication_enable = `OFF;
1371            end
1372            R_MOD_0: begin
1373                mod_p_select = 1'b0;
1374                alu_in1 = r;
1375                alu_in2 = {ORDER{1'b0}};
1376                op = ADD;
1377                alu_enable = `ON;
1378                alu_start = `HIGH;
1379            end
1380            R_MOD_2: begin
1381                r = alu_out_w;
1382                alu_enable = `OFF;
1383                alu_start = `LOW;
1384            end
```

```
1385            S_MUL_0: begin
1386                alu_enable = 'ON;
1387                alu_in1 = a;
1388                alu_in2 = r;
1389                op = MUL;
1390                alu_start = 'HIGH;
1391            end
1392            S_MUL_2: begin
1393                s = alu_out_w;
1394                alu_start = 'LOW;
1395            end
1396            S_ADD_0: begin
1397                alu_in1 = z;
1398                alu_in2 = s;
1399                op = ADD;
1400                alu_start = 'HIGH;
1401            end
1402            S_ADD_2: begin
1403                s = alu_out_w;
1404                alu_start = 'LOW;
1405            end
1406            S_INV_0: begin
1407                alu_in1 = k;
1408                op = MOD_INV;
1409                alu_start = 'HIGH;
1410            end
1411            S_INV_2: begin
1412                alu_in2 = alu_out_w;
1413                alu_start = 'LOW;
1414            end
1415            S_PROD_0: begin
1416                alu_in1 = s;
1417                op = MUL;
1418                alu_start = 'HIGH;
1419            end
1420            S_PROD_2: begin
1421                s = alu_out_w;
1422                alu_start = 'LOW;
1423            end
1424            EXIT: begin
1425                done = 'HIGH;
1426                alu_enable = 'LOW;
1427            end
1428        endcase
1429    end
1430
1431
1432    assign mod_p_w = (mod_p_select) ? p : n;
1433    assign alu_enable_w = (alu_enable) | (slope_enable) | (
        point_enable);
1434    assign alu_start_w = (alu_enable == 'ON) ? alu_start : (
        slope_enable == 'ON) ? slope_start : (point_enable == 'ON) ?
        point_start : 1'hz;
1435    assign in1_w = (alu_enable == 'ON) ? alu_in1 : (slope_enable ==
        'ON) ? slope_in1 : (point_enable == 'ON) ? point_in1 : {ORDER
        {1'hz}};
```

```
1436      assign in2_w = (alu_enable == `ON) ? alu_in2 : (slope_enable ==
              `ON) ? slope_in2 : (point_enable == `ON) ? point_in2 : {ORDER
              {1'hz}};
1437      assign op_w = (alu_enable == `ON) ? op : (slope_enable == `ON) ?
              slope_op : (point_enable == `ON) ? point_op : 1'hz;
1438
1439
1440  endmodule
1441
1442  module ecdsa_top(
1443      input clk,
1444      input ready,
1445      output done,
1446      input [ORDER-1:0] z,
1447      output [ORDER-1:0] r,
1448      output [ORDER-1:0] s
1449      );
1450      reg [ORDER-1:0] prime;
1451      wire enable, start;
1452      reg [ORDER-1:0] G_x, G_y;
1453      reg [ORDER-1:0] p,a,k, n;
1454      reg [ORDER-1:0] curve_a;
1455      assign enable = ready ? 1'b1 : 1'b0;;
1456      assign start = ready ? 1'b1: 1'b0;
1457      ecdsa_main x1(
1458          .clk(clk),
1459          .enable(enable),
1460          .start(start),
1461          .G_x(G_x), .G_y(G_y),
1462          .z(z), .p(p), .k(k), .a(a), .n(n),
1463          .curve_a(curve_a),
1464          .done(done), .r(r), .s(s)
1465      );
1466
1467      always_comb begin
1468          p = 256'
                  hfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f
                  ;
1469          n = 256'
                  hfffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141
                  ;
1470          a = 256'h2ab;
1471          curve_a = 0;
1472          G_x = 256'
                  h79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
                  ;
1473          G_y = 256'
                  h483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
                  ;
1474          k = 19;
1475      end
1476  endmodule
```

A.4 SHA256

A.4.1 SHA Top

```
1  module SHA_top(
2            input wire clk ,
3            input wire reset ,
4            input wire enable ,
5            output wire sha_digest_valid ,
6            output wire [255:0] sha256_hash_data
7       );
8
9  parameter blocks = 1 ;
10 reg next_block ;
11 reg [9:0] count ;
12 reg [511 : 0] block_SHA ;
13 reg initial_block  ;
14 reg sha_valid = 0 ;
15 reg [255 : 0] Hash_data ;
16 wire [511 : 0]  block_data_1 = 512'
     h9a35b0e2435bd74c41e98b4440c6d91742187dd900000000449a1bf80000000042707bb308050
      ;
17 wire [511 : 0]  block_data_2 = 512'
     h0000000000000000000000000000000000000000000000000000000000000000000000000000
      ;
18 wire Hash_digest_valid ;
19 reg Hash_digest_valid_1 ;
20 reg Hash_digest_valid_2 ;
21 wire SHA_ready ;
22 wire [255 : 0] SHA_digest ;
23
24 sha256 S1(
25              .clk(clk),
26              .reset_n(reset),
27              .init(initial_block),
28              .next(next_block),
29              .block(block_SHA),
30              .ready(SHA_ready),
31              .digest(SHA_digest),
32              .digest_valid(Hash_digest_valid)
33          );
34
35
36    always @ (posedge clk or posedge reset) begin
37       if(reset == 1)
38       begin
39          next_block <= 0 ;
40          count <= 0 ;
41          initial_block <= 0 ;
42          Hash_data <= 0 ;
43          block_SHA <= 0 ;
44          sha_valid <= 0 ;
45       end
46       else if(enable == 1)
47       begin
```

```
48          Hash_digest_valid_1 <= Hash_digest_valid ;
49          Hash_digest_valid_2 <= Hash_digest_valid_1 ;
50          if(Hash_digest_valid == 1'b0 && SHA_ready == 1'b1 && count
               == 0)
51          begin
52             initial_block <= 1 ;
53             block_SHA <= block_data_1 ;
54             count <= count + 1 ;
55             next_block <= 0 ;
56          end
57          else if(Hash_digest_valid == 1'b1 && SHA_ready == 1'b1 &&
               count >= 1 && count < blocks )
58          begin
59             initial_block <= 0 ;
60             next_block <= 1 ;
61             count <= count + 1 ;
62             block_SHA <= block_data_2 ;
63          end
64          else if(Hash_digest_valid == 1'b1 && Hash_digest_valid_1 ==
               1'b1 && Hash_digest_valid_2 == 1'b1 && SHA_ready == 1'b1
               && count == blocks)
65          begin
66             sha_valid <= 1 ;
67          end
68          else
69          begin
70             initial_block <= 0 ;
71             next_block <= 0 ;
72          end
73
74          if(Hash_digest_valid == 1'b1)
75          begin
76             Hash_data <= SHA_digest ;
77          end
78       end
79    end
80
81    assign sha_digest_valid = sha_valid ;
82    assign sha256_hash_data = Hash_data ;
83 endmodule
```

A.4.2 SHA256

```
1
2  module sha256(
3                     input wire             clk,
4                     input wire             reset_n,
5                     input wire             init,
6                     input wire             next,
7                     input wire [511 : 0]   block,
8                     output wire            ready,
9                     output wire [255 : 0]  digest,
10                    output wire            digest_valid
11                   );
12
13   wire [31:0] initial_SHA256_H0 = 32'h6a09e667;
```

```
14    wire [31:0] initial_SHA256_H1 = 32'hbb67ae85;
15    wire [31:0] initial_SHA256_H2 = 32'h3c6ef372;
16    wire [31:0] initial_SHA256_H3 = 32'ha54ff53a;
17    wire [31:0] initial_SHA256_H4 = 32'h510e527f;
18    wire [31:0] initial_SHA256_H5 = 32'h9b05688c;
19    wire [31:0] initial_SHA256_H6 = 32'h1f83d9ab;
20    wire [31:0] initial_SHA256_H7 = 32'h5be0cd19;
21
22    localparam CTRL_IDLE   = 0;
23    localparam CTRL_ROUNDS = 1;
24    localparam CTRL_DONE   = 2;
25
26    reg [31 : 0] a_reg;
27    reg [31 : 0] b_reg;
28    reg [31 : 0] c_reg;
29    reg [31 : 0] d_reg;
30    reg [31 : 0] e_reg;
31    reg [31 : 0] f_reg;
32    reg [31 : 0] g_reg;
33    reg [31 : 0] h_reg;
34
35    reg [31 : 0] H0_reg;
36    reg [31 : 0] H0_new;
37    reg [31 : 0] H1_reg;
38    reg [31 : 0] H1_new;
39    reg [31 : 0] H2_reg;
40    reg [31 : 0] H2_new;
41    reg [31 : 0] H3_reg;
42    reg [31 : 0] H3_new;
43    reg [31 : 0] H4_reg;
44    reg [31 : 0] H4_new;
45    reg [31 : 0] H5_reg;
46    reg [31 : 0] H5_new;
47    reg [31 : 0] H6_reg;
48    reg [31 : 0] H6_new;
49    reg [31 : 0] H7_reg;
50    reg [31 : 0] H7_new;
51    reg          H_we;
52
53    reg [5 : 0] counter_pos;
54
55    reg digest_valid_reg;
56    reg digest_valid_new;
57    reg digest_valid_we;
58
59    reg [1 : 0] sha256_ctrl_reg;
60    reg [1 : 0] sha256_ctrl_new;
61    reg         sha256_ctrl_we;
62
63    reg digest_init;
64    reg digest_update;
65    reg state_update;
66    reg ready_flag;
67
68    reg [31 : 0] t1;
69    reg [31 : 0] t2;
```

```
70
71    wire [31 : 0] k_data;
72    wire [31 : 0] w_data;
73
74    wire constant_pos = (init || next)  ;
75    wire next_word = (sha256_ctrl_reg == CTRL_ROUNDS)?1:0 ;
76
77    sha256_constants k_constants_inst(
78                                      .position(counter_pos),
79                                      .constant_value(k_data)
80                                     );
81
82    sha256_w_mem w_mem_inst(
83                            .clk(clk),
84                            .reset_n(reset_n),
85                            .block(block),
86                            .init(constant_pos),
87                            .next(next_word),
88                            .w(w_data)
89                           );
90
91    assign ready = ready_flag;
92
93    assign digest = {H0_reg, H1_reg, H2_reg, H3_reg,
94                     H4_reg, H5_reg, H6_reg, H7_reg};
95
96    assign digest_valid = digest_valid_reg;
97
98    always @ (posedge clk or posedge reset_n)
99      begin : reg_update
100       if (reset_n)
101         begin
102           H0_reg           <= 32'h0;
103           H1_reg           <= 32'h0;
104           H2_reg           <= 32'h0;
105           H3_reg           <= 32'h0;
106           H4_reg           <= 32'h0;
107           H5_reg           <= 32'h0;
108           H6_reg           <= 32'h0;
109           H7_reg           <= 32'h0;
110           digest_valid_reg <= 0;
111           sha256_ctrl_reg  <= CTRL_IDLE;
112         end
113       else
114         begin
115           if (H_we)
116             begin
117               H0_reg <= H0_new;
118               H1_reg <= H1_new;
119               H2_reg <= H2_new;
120               H3_reg <= H3_new;
121               H4_reg <= H4_new;
122               H5_reg <= H5_new;
123               H6_reg <= H6_new;
124               H7_reg <= H7_new;
125             end
```

```
126
127             if (digest_valid_we)
128               digest_valid_reg <= digest_valid_new;
129             if (sha256_ctrl_we)
130               sha256_ctrl_reg <= sha256_ctrl_new;
131           end
132       end
133
134     always @(posedge clk or posedge reset_n )
135     begin : counter
136       if(reset_n||next||init)
137         begin
138           counter_pos <= 0 ;
139         end
140       else if(sha256_ctrl_reg == CTRL_ROUNDS)
141         begin
142           counter_pos <= counter_pos + 1 ;
143         end
144     end
145
146     always @*
147       begin : digest_logic
148         H0_new = 32'h0;
149         H1_new = 32'h0;
150         H2_new = 32'h0;
151         H3_new = 32'h0;
152         H4_new = 32'h0;
153         H5_new = 32'h0;
154         H6_new = 32'h0;
155         H7_new = 32'h0;
156         H_we = 0;
157
158         if (digest_init)
159           begin
160             H_we = 1;
161             H0_new = initial_SHA256_H0;
162             H1_new = initial_SHA256_H1;
163             H2_new = initial_SHA256_H2;
164             H3_new = initial_SHA256_H3;
165             H4_new = initial_SHA256_H4;
166             H5_new = initial_SHA256_H5;
167             H6_new = initial_SHA256_H6;
168             H7_new = initial_SHA256_H7;
169           end
170
171         if (digest_update)
172           begin
173             H0_new = H0_reg + a_reg;
174             H1_new = H1_reg + b_reg;
175             H2_new = H2_reg + c_reg;
176             H3_new = H3_reg + d_reg;
177             H4_new = H4_reg + e_reg;
178             H5_new = H5_reg + f_reg;
179             H6_new = H6_reg + g_reg;
180             H7_new = H7_reg + h_reg;
181             H_we = 1;
```

```
182         end
183       end
184
185     always @*
186       begin : t1_logic
187         reg [31 : 0] sum1;
188         reg [31 : 0] ch;
189
190         sum1 = {e_reg[5  : 0], e_reg[31 :  6]} ^
191                {e_reg[10 : 0], e_reg[31 : 11]} ^
192                {e_reg[24 : 0], e_reg[31 : 25]};
193
194         ch = (e_reg & f_reg) ^ ((~e_reg) & g_reg);
195
196         t1 = h_reg + sum1 + ch + w_data + k_data;
197       end
198
199
200     always @*
201       begin : t2_logic
202         reg [31 : 0] sum0;
203         reg [31 : 0] maj;
204
205         sum0 = {a_reg[1  : 0], a_reg[31 :  2]} ^
206                {a_reg[12 : 0], a_reg[31 : 13]} ^
207                {a_reg[21 : 0], a_reg[31 : 22]};
208
209         maj = (a_reg & b_reg) ^ (a_reg & c_reg) ^ (b_reg & c_reg);
210
211         t2 = sum0 + maj;
212       end
213
214     always @(posedge clk or posedge reset_n)
215       begin : working_registers
216         if(reset_n)
217           begin
218             a_reg             <= 32'h0;
219             b_reg             <= 32'h0;
220             c_reg             <= 32'h0;
221             d_reg             <= 32'h0;
222             e_reg             <= 32'h0;
223             f_reg             <= 32'h0;
224             g_reg             <= 32'h0;
225             h_reg             <= 32'h0;
226           end
227         else if(init)
228           begin
229             a_reg  <= initial_SHA256_H0;
230             b_reg  <= initial_SHA256_H1;
231             c_reg  <= initial_SHA256_H2;
232             d_reg  <= initial_SHA256_H3;
233             e_reg  <= initial_SHA256_H4;
234             f_reg  <= initial_SHA256_H5;
235             g_reg  <= initial_SHA256_H6;
236             h_reg  <= initial_SHA256_H7;
237           end
```

```
238         else if(next)
239           begin
240             a_reg  <= H0_reg;
241             b_reg  <= H1_reg;
242             c_reg  <= H2_reg;
243             d_reg  <= H3_reg;
244             e_reg  <= H4_reg;
245             f_reg  <= H5_reg;
246             g_reg  <= H6_reg;
247             h_reg  <= H7_reg;
248           end
249         else if(sha256_ctrl_reg == CTRL_ROUNDS)
250           begin
251             a_reg  <= t1 + t2;
252             b_reg  <= a_reg;
253             c_reg  <= b_reg;
254             d_reg  <= c_reg;
255             e_reg  <= d_reg + t1;
256             f_reg  <= e_reg;
257             g_reg  <= f_reg;
258             h_reg  <= g_reg;
259           end
260       end
261
262     always @*
263       begin : sha256_ctrl_fsm
264         digest_init      = 0;
265         digest_update    = 0;
266
267         state_update     = 0;
268
269         ready_flag       = 0;
270
271         digest_valid_new = 0;
272         digest_valid_we  = 0;
273
274         sha256_ctrl_new  = CTRL_IDLE;
275         sha256_ctrl_we   = 0;
276
277
278         case (sha256_ctrl_reg)
279           CTRL_IDLE:
280             begin
281               ready_flag = 1;
282
283               if (init)
284                 begin
285                   digest_init      = 1;
286                   digest_valid_new = 0;
287                   digest_valid_we  = 1;
288                   sha256_ctrl_new  = CTRL_ROUNDS;
289                   sha256_ctrl_we   = 1;
290                 end
291
292               if (next)
293                 begin
```

```
294                 digest_valid_new = 0;
295                 digest_valid_we  = 1;
296                 sha256_ctrl_new  = CTRL_ROUNDS;
297                 sha256_ctrl_we   = 1;
298               end
299           end
300
301         CTRL_ROUNDS:
302           begin
303             state_update = 1;
304
305             if (counter_pos == 63)
306               begin
307                 sha256_ctrl_new = CTRL_DONE;
308                 sha256_ctrl_we  = 1;
309               end
310           end
311
312         CTRL_DONE:
313           begin
314             digest_update    = 1;
315             digest_valid_new = 1;
316             digest_valid_we  = 1;
317             sha256_ctrl_new  = CTRL_IDLE;
318             sha256_ctrl_we   = 1;
319           end
320       endcase
321     end
322
323 endmodule
```

A.4.3 SHA Constants

```
1  module sha256_constants(
2                          input wire  [5 : 0] position,
3                          output reg [31 : 0] constant_value
4                         );
5
6    always @(*)
7      begin : constant_mem
8        case(position)
9          00: constant_value = 32'h428a2f98;
10         01: constant_value = 32'h71374491;
11         02: constant_value = 32'hb5c0fbcf;
12         03: constant_value = 32'he9b5dba5;
13         04: constant_value = 32'h3956c25b;
14         05: constant_value = 32'h59f111f1;
15         06: constant_value = 32'h923f82a4;
16         07: constant_value = 32'hab1c5ed5;
17         08: constant_value = 32'hd807aa98;
18         09: constant_value = 32'h12835b01;
19         10: constant_value = 32'h243185be;
20         11: constant_value = 32'h550c7dc3;
21         12: constant_value = 32'h72be5d74;
22         13: constant_value = 32'h80deb1fe;
23         14: constant_value = 32'h9bdc06a7;
```

```
24            15: constant_value = 32'hc19bf174;
25            16: constant_value = 32'he49b69c1;
26            17: constant_value = 32'hefbe4786;
27            18: constant_value = 32'h0fc19dc6;
28            19: constant_value = 32'h240ca1cc;
29            20: constant_value = 32'h2de92c6f;
30            21: constant_value = 32'h4a7484aa;
31            22: constant_value = 32'h5cb0a9dc;
32            23: constant_value = 32'h76f988da;
33            24: constant_value = 32'h983e5152;
34            25: constant_value = 32'ha831c66d;
35            26: constant_value = 32'hb00327c8;
36            27: constant_value = 32'hbf597fc7;
37            28: constant_value = 32'hc6e00bf3;
38            29: constant_value = 32'hd5a79147;
39            30: constant_value = 32'h06ca6351;
40            31: constant_value = 32'h14292967;
41            32: constant_value = 32'h27b70a85;
42            33: constant_value = 32'h2e1b2138;
43            34: constant_value = 32'h4d2c6dfc;
44            35: constant_value = 32'h53380d13;
45            36: constant_value = 32'h650a7354;
46            37: constant_value = 32'h766a0abb;
47            38: constant_value = 32'h81c2c92e;
48            39: constant_value = 32'h92722c85;
49            40: constant_value = 32'ha2bfe8a1;
50            41: constant_value = 32'ha81a664b;
51            42: constant_value = 32'hc24b8b70;
52            43: constant_value = 32'hc76c51a3;
53            44: constant_value = 32'hd192e819;
54            45: constant_value = 32'hd6990624;
55            46: constant_value = 32'hf40e3585;
56            47: constant_value = 32'h106aa070;
57            48: constant_value = 32'h19a4c116;
58            49: constant_value = 32'h1e376c08;
59            50: constant_value = 32'h2748774c;
60            51: constant_value = 32'h34b0bcb5;
61            52: constant_value = 32'h391c0cb3;
62            53: constant_value = 32'h4ed8aa4a;
63            54: constant_value = 32'h5b9cca4f;
64            55: constant_value = 32'h682e6ff3;
65            56: constant_value = 32'h748f82ee;
66            57: constant_value = 32'h78a5636f;
67            58: constant_value = 32'h84c87814;
68            59: constant_value = 32'h8cc70208;
69            60: constant_value = 32'h90befffa;
70            61: constant_value = 32'ha4506ceb;
71            62: constant_value = 32'hbef9a3f7;
72            63: constant_value = 32'hc67178f2;
73          endcase
74        end
75
76  endmodule
```

A.4.4 SHA256 W Mem

```
1
2  module sha256_w_mem(
3                      input wire            clk,
4                      input wire            reset_n,
5
6                      input wire [511 : 0] block,
7
8                      input wire            init,
9                      input wire            next,
10                     output wire [31 : 0] w
11                    );
12
13   reg [31 : 0] w_mem [0 : 15];
14   reg [31 : 0] w_mem00_new;
15   reg [31 : 0] w_mem01_new;
16   reg [31 : 0] w_mem02_new;
17   reg [31 : 0] w_mem03_new;
18   reg [31 : 0] w_mem04_new;
19   reg [31 : 0] w_mem05_new;
20   reg [31 : 0] w_mem06_new;
21   reg [31 : 0] w_mem07_new;
22   reg [31 : 0] w_mem08_new;
23   reg [31 : 0] w_mem09_new;
24   reg [31 : 0] w_mem10_new;
25   reg [31 : 0] w_mem11_new;
26   reg [31 : 0] w_mem12_new;
27   reg [31 : 0] w_mem13_new;
28   reg [31 : 0] w_mem14_new;
29   reg [31 : 0] w_mem15_new;
30   reg          w_mem_we;
31
32   reg [5 : 0] w_ctr_reg;
33   reg [5 : 0] w_ctr_new;
34   reg         w_ctr_we;
35
36   reg [31 : 0] w_tmp;
37   reg [31 : 0] w_new;
38
39   assign w = w_tmp;
40
41   always @ (posedge clk or posedge reset_n)
42     begin : reg_update
43       integer i;
44
45       if (reset_n)
46         begin
47           for (i = 0 ; i < 16 ; i = i + 1)
48             w_mem[i] <= 32'h0;
49
50           w_ctr_reg <= 6'h0;
51         end
52       else
53         begin
54           if (w_mem_we)
55             begin
56               w_mem[00] <= w_mem00_new;
```

```
57                  w_mem[01] <= w_mem01_new;
58                  w_mem[02] <= w_mem02_new;
59                  w_mem[03] <= w_mem03_new;
60                  w_mem[04] <= w_mem04_new;
61                  w_mem[05] <= w_mem05_new;
62                  w_mem[06] <= w_mem06_new;
63                  w_mem[07] <= w_mem07_new;
64                  w_mem[08] <= w_mem08_new;
65                  w_mem[09] <= w_mem09_new;
66                  w_mem[10] <= w_mem10_new;
67                  w_mem[11] <= w_mem11_new;
68                  w_mem[12] <= w_mem12_new;
69                  w_mem[13] <= w_mem13_new;
70                  w_mem[14] <= w_mem14_new;
71                  w_mem[15] <= w_mem15_new;
72                end
73
74              if (w_ctr_we)
75                w_ctr_reg <= w_ctr_new;
76            end
77        end
78
79      always @*
80        begin : select_w
81          if (w_ctr_reg < 16)
82            w_tmp = w_mem[w_ctr_reg[3 : 0]];
83          else
84            w_tmp = w_new;
85        end
86
87      always @*
88        begin : w_mem_update_logic
89          reg [31 : 0] w_0;
90          reg [31 : 0] w_1;
91          reg [31 : 0] w_9;
92          reg [31 : 0] w_14;
93          reg [31 : 0] d0;
94          reg [31 : 0] d1;
95
96          w_mem00_new = 32'h0;
97          w_mem01_new = 32'h0;
98          w_mem02_new = 32'h0;
99          w_mem03_new = 32'h0;
100         w_mem04_new = 32'h0;
101         w_mem05_new = 32'h0;
102         w_mem06_new = 32'h0;
103         w_mem07_new = 32'h0;
104         w_mem08_new = 32'h0;
105         w_mem09_new = 32'h0;
106         w_mem10_new = 32'h0;
107         w_mem11_new = 32'h0;
108         w_mem12_new = 32'h0;
109         w_mem13_new = 32'h0;
110         w_mem14_new = 32'h0;
111         w_mem15_new = 32'h0;
112         w_mem_we    = 0;
```

```
113
114         w_0  = w_mem[0];
115         w_1  = w_mem[1];
116         w_9  = w_mem[9];
117         w_14 = w_mem[14];
118
119         d0 = {w_1[6  : 0], w_1[31 :  7]} ^
120              {w_1[17 : 0], w_1[31 : 18]} ^
121              {3'b000, w_1[31 : 3]};
122
123         d1 = {w_14[16 : 0], w_14[31 : 17]} ^
124              {w_14[18 : 0], w_14[31 : 19]} ^
125              {10'b0000000000, w_14[31 : 10]};
126
127         w_new = d1 + w_9 + d0 + w_0;
128
129         if (init)
130           begin
131             w_mem00_new = block[511 : 480];
132             w_mem01_new = block[479 : 448];
133             w_mem02_new = block[447 : 416];
134             w_mem03_new = block[415 : 384];
135             w_mem04_new = block[383 : 352];
136             w_mem05_new = block[351 : 320];
137             w_mem06_new = block[319 : 288];
138             w_mem07_new = block[287 : 256];
139             w_mem08_new = block[255 : 224];
140             w_mem09_new = block[223 : 192];
141             w_mem10_new = block[191 : 160];
142             w_mem11_new = block[159 : 128];
143             w_mem12_new = block[127 :  96];
144             w_mem13_new = block[95  :  64];
145             w_mem14_new = block[63  :  32];
146             w_mem15_new = block[31  :   0];
147             w_mem_we    = 1;
148           end
149
150         if (next && (w_ctr_reg > 15))
151           begin
152             w_mem00_new = w_mem[01];
153             w_mem01_new = w_mem[02];
154             w_mem02_new = w_mem[03];
155             w_mem03_new = w_mem[04];
156             w_mem04_new = w_mem[05];
157             w_mem05_new = w_mem[06];
158             w_mem06_new = w_mem[07];
159             w_mem07_new = w_mem[08];
160             w_mem08_new = w_mem[09];
161             w_mem09_new = w_mem[10];
162             w_mem10_new = w_mem[11];
163             w_mem11_new = w_mem[12];
164             w_mem12_new = w_mem[13];
165             w_mem13_new = w_mem[14];
166             w_mem14_new = w_mem[15];
167             w_mem15_new = w_new;
168             w_mem_we    = 1;
```

```
169           end
170       end
171
172     always @*
173       begin : w_ctr
174         w_ctr_new = 6'h0;
175         w_ctr_we  = 1'h0;
176
177         if (init)
178           begin
179             w_ctr_new = 6'h0;
180             w_ctr_we  = 1'h1;
181           end
182
183         if (next)
184           begin
185             w_ctr_new = w_ctr_reg + 6'h01;
186             w_ctr_we  = 1'h1;
187           end
188       end
189 endmodule
```

A.5 UART

A.5.1 UART Top

```
1  module UART_Top(
2  input clk,
3  input reset,
4  input transmit,
5  output TxD
6  );
7
8  wire transmit_out;
9  wire tranmit_and_valid ;
10 wire valid = 1'b1 ;
11 wire [511:0] data_512 ;
12 wire [255:0] data = 255'
       h514D41524A49545241414A414D41524A49545241414A414D41524A4954524141
       ;
13
14 hex_to_ascii CV ( data, data_512) ;
15 Debounce_Signals DB (clk, transmit, transmit_out);
16 Transmitter T1 (clk, reset, tranmit_and_valid, data_512, TxD);
17
18 assign tranmit_and_valid = valid && transmit_out;
19
20 endmodule
```

A.5.2 Hex to ASCII

```
1  module hex_to_ascii(
2      input [255:0] data,
3      output [511:0] out_data
4  );
```

```
5
6  integer i ;
7  reg [3:0] in_data ;
8  reg [7:0] ascii_data ;
9  reg [511:0] ascii_con ;
10
11 always @(*)
12 begin
13     for ( i = 0; i< 64 ; i = i+1 ) begin
14         in_data = {data[i*4+3],data[i*4+2],data[i*4+1],data[i*4]} ;
15         case (in_data)
16             4'h0: ascii_data = 8'h30;
17             4'h1: ascii_data = 8'h31;
18             4'h2: ascii_data = 8'h32;
19             4'h3: ascii_data = 8'h33;
20             4'h4: ascii_data = 8'h34;
21             4'h5: ascii_data = 8'h35;
22             4'h6: ascii_data = 8'h36;
23             4'h7: ascii_data = 8'h37;
24             4'h8: ascii_data = 8'h38;
25             4'h9: ascii_data = 8'h39;
26             4'hA: ascii_data = 8'h41;
27             4'hB: ascii_data = 8'h42;
28             4'hC: ascii_data = 8'h43;
29             4'hD: ascii_data = 8'h44;
30             4'hE: ascii_data = 8'h45;
31             4'hF: ascii_data = 8'h46;
32         endcase
33         {ascii_con[i*8+7],ascii_con[i*8+6],ascii_con[i*8+5],
34         ascii_con[i*8+4],ascii_con[i*8+3],ascii_con[i*8+2],ascii_con[
               i*8+1],ascii_con[i*8]} = ascii_data ;
35     end
36 end
37
38 assign out_data = ascii_con ;
39
40 endmodule
```

A.5.3 Debounce Signal

```
1  module Debounce_Signals #(parameter threshold = 100000)
2  (
3  input clk,
4  input btn1,
5  output reg transmit
6      );
7
8  reg button_ff1 = 0;
9  reg button_ff2 = 0;
10 reg [30:0]count = 0;
11
12 always @(posedge clk)begin
13 button_ff1 <= btn1;
14 button_ff2 <= button_ff1;
15 end
16
```

```
always @(posedge clk) begin
 if (button_ff2)
 begin
    if (~&count)
        count <= count+1;
 end else begin
    if (|count)
        count <= count-1;
 end
 if (count > threshold)
    transmit <= 1;
 else
    transmit <= 0;
end


endmodule
```

A.5.4 Transmitter

```
module Transmitter(
input clk,
input reset,
input transmit,
input [511:0] in_data,
output reg TxD
    );

reg [3:0] bitcounter;
reg [5:0] total_data_counter = 6'b111111 ;
reg [13:0] counter;
reg [1:0] state,nextstate;
reg [9:0] rightshiftreg;
reg shift;
reg load;
reg clear;
reg [7:0] data = 8'h41  ;

integer i,j ;

always @ (posedge clk)
begin
    if (reset)
            begin
        state <=0 ;
        counter <=0 ;
        bitcounter <=0 ;
        total_data_counter <= 6'b111111 ;
       end
    else begin
         counter <= counter + 1;
            if (counter >= 10415)
               begin
                  state <= nextstate;
                  counter <= 0;
                  if (load)
```

```
37                      begin
38                        total_data_counter <= total_data_counter - 1 ;
39                        rightshiftreg <= {1'b1,in_data[total_data_counter
                              *8+7],in_data[total_data_counter*8+6],in_data[
                              total_data_counter*8+5],in_data[
                              total_data_counter*8+4],
40                        in_data[total_data_counter*8+3],in_data[
                              total_data_counter*8+2],in_data[
                              total_data_counter*8+1],in_data[
                              total_data_counter*8],1'b0};
41                      end
42                              if (clear) bitcounter <=0;
43                      if (shift)
44                         begin
45                            rightshiftreg <= rightshiftreg >> 1;
46                            bitcounter <= bitcounter + 1;
47                         end
48                  end
49           end
50  end
51
52  always @ (posedge clk)
53
54  begin
55      load <=0;
56      shift <=0;
57      clear <=0;
58      TxD <=1;
59      case (state)
60          0: begin
61              if (transmit) begin
62              nextstate <= 1;
63              load <=1;
64              shift <=0;
65              clear <=0;
66              end
67                  else begin
68              nextstate <= 0;
69              TxD <= 1;
70              end
71             end
72          1: begin
73              if (bitcounter >=10) begin
74              nextstate <= 2;
75              clear <=1;
76              end
77                  else begin
78              nextstate <= 1;
79              TxD <= rightshiftreg[0];
80              shift <=1;
81              end
82             end
83          2: begin
84              if(total_data_counter == 0)
85              begin
86                  nextstate <= 0 ;
```

```
87                      clear <= 1 ;
88                  end
89                  else
90                  begin
91                      nextstate <= 1 ;
92                      load <= 1 ;
93                      shift <= 0 ;
94                      clear <= 0 ;
95                  end
96              end
97               default: nextstate <= 0;
98          endcase
99  end
100
101 endmodule
```