RSA Encryption and Decryption Implementation in an FPGA Using Verilog HDL

Dr. Mohammad H. Awedh[1*]

Associate Professor, Department of Electrical and Computer Engineering, Faculty of Engineering, King Abdulaziz University, Saudi Arabia[1]

Email: mhawedh@kau.edu.sa

Dr. Ahmed Mueen[2]

Associate Professor, Department of Computer and Information Technology, King Abdulaziz University, Saudi Arabia [2]

Email: mueen@kau.edu.sa

Abstract:

The main objective of this project is to create a hardware-based system that is capable of encrypting plaintext and decrypting ciphertext when the public or private key and modulus n are given. Normally, the RSA algorithm has performance limitations in software-based realizations due to its computational difficulty in prime factorization. Our proposed design addresses this issue by utilizing the parallel processing capabilities of FPGA architectures. For efficient cryptographic processing, we have developed and optimized components such as modular multiplication and modular exponentiation using Verilog HDL. The results show that FPGA-based implementations are more suitable for secure real-time and embedded cryptographic applications as compared to traditional software approaches. Based on the result of this research the authors recommend to focus on optimizing cryptographic hardware for power efficiency and scalability, especially for use in IoT and edge devices, and encouraged to explore the integration of RSA hardware modules with other security protocols for comprehensive system-on-chip (SoC) solutions, also recommend the educational institutions and training programs to incorporate practical FPGA-based cryptographic projects to bridge the gap between theory and application and continuing to refine RSA hardware implementations and exploring new architectural strategies.

Keywords: Cryptography, RSA algorithm, FPGA architecture, Real-time and Embedded cryptographic, Hardware Description Language, UART.

1. Introduction

In modern secure communication of information systems, cryptography, the science of encoding information to protect it from unauthorized access, plays a vital role in ensuring data integrity, confidentiality, authentication, and non-repudiation. One of the most robustness and mathematical soundness cryptographic algorithms is RSA (Rivest–Shamir–Adleman) (Ugbedeojo, Adebiyi, Aroba, & Adebiyi, 2024, pp. 1–27; Vidhyalakshmi & Ramesh, 2013). RSA algorithm is introduced in 1977, it employs asymmetric key cryptography, utilizing a pair of mathematically linked keys—one public for encryption and one private for decryption. RSA encryption offers a secure method for protecting sensitive information with its computational difficulty of factoring the product of two large prime numbers. This foundation remains an effective despite recent vulnerabilities in implementations using smaller key sizes (Zheng, 2024; Feng, Nitaj, & Pan, 2024). RSA particularly suitable for secure key exchange and digital signatures. RSA is computationally intensive, especially for large key sizes (Zheng, 2024). Traditional software implementations often face performance bottlenecks, limiting their utility in real-time or resource-constrained environments. To address these limitations, hardware-based approaches (using FPGAs) offer significant advantages. FPGAs provide performance efficiency, parallelism and re-configurability, making them ideal for accelerating cryptographic algorithms and increase their performance (Zhang & Zhang, 2012).

In this paper, we present the design and implementation of an RSA encryption/decryption system on an FPGA using Verilog HDL. The primary objective is to develop a high-performance, hardware-optimized cryptographic engine capable of securely processing data in real-time. By focusing on the implementation of modular multiplication and modular exponentiation, the core operations in RSA, our work demonstrates the effectiveness of FPGA platforms in handling the computational demands of public-key cryptography.

2. Related Work

Significant research efforts have been made on the RSA encryption algorithm using Verilog HDL on FPGA platforms for hardware implementation. Addressing different aspects of the research, such as performance optimization, architectural innovation, and design challenges. An encryption engine was generated in study of Vidhyalakshmi and Ramesh (2013), focusing on key creation, encryption, and decryption processes. The method the researchers used is right-to-left binary exponentiation and utilizes a primality tester for random prime number generation.

They used Verilog coding for the implementation and synthesized using the Xilinx 13.2 design suite. The results show that it is an efficient approach to RSA encryption at moderate key sizes. The design and verification of an RSA encryption system were identified in study of Zhang and Zhang (2012). The main goal of the research was to improve modular operation efficiency. The study emphasized improving modular operation efficiency through Montgomery modular multiplication.. The method and design researchers introduced have successfully decreased computational complexity. It makes the system very suitable for real-time cryptographic applications on FPGA platforms.

The work by Alshahrani (2017) also focused on implementing RSA using Verilog HDL. The complete details have been given for key generation, encryption, and decryption. The study also provided valuable details about the challenges often experienced during hardware implementation phases.

In study of Arun and Dharani (2020), the Shift-Sub Modular Multiplication (SSMM) algorithm was introduced. The research presents a division-free approach to modular multiplication. The RSA cryptosystem was implemented using Verilog HDL. The result of the performance comparison shows efficient gains in specific operational contexts by traditional Montgomery multiplication techniques. A different mathematical approach was presented in study of Raut and Raut (2006). The method from Ancient Indian Vedic Mathematics was employed to design the RSA cryptosystem. This research work utilized a hierarchical overlay multiplier architecture and a Straight Division algorithm. The system decreases hardware complexity and increases performance when synthesized on Xilinx Spartan FPGAs. Using larger key sizes by expanding the RSA implementation is proposed (Abid & Khan, 2019). The design is an RSA encryption incorporating Montgomery modular multiplication using Verilog HDL, which improved efficiency. The research provided a comprehensive hardware implementation and simulation. The proposed approach shows the effectiveness when security is important.

A comparative study was conducted by Kochte et al. (2016), which analyzed different FPGA-based architectures for RSA implementation. The study examines different features of concurrency and sequential operation designs. This research work discussed in detail resource utilization and power consumption. Research assists designers in selecting appropriate architectures based on system requirements.

Saini (2017) presented an efficient primality testing algorithm optimized for 64-bit RSA encryption. They have used the right-to-left binary exponentiation method. The approach is suitable for constrained hardware environments by providing enhanced performance while minimizing FPGA area usage. The study by Kumar and Sharma (2015) utilized Montgomery multiplication with Block RAM (BRAM) blocks to store operands. The design achieved a suitable and efficient RSA operation by avoiding hard-wired modulus values and dynamically managing operands.

The above-mentioned research work collectively explains the different techniques and designs. Studies investigated hardware implementation of RSA encryption by highlighting improvements in speed, resource utilization, scalability, and overall system efficiency.

3. RSA Cryptographic Communication Model

The RSA Cryptographic Communication Model is a framework that explains how the RSA cryptographic algorithm is used to ensure secure communication between two parties over an insecure channel.

Figure 1 illustrates the fundamental working of the RSA public-key cryptographic algorithm. The process starts when the sender wants to send a confidential message. This message is in readable form, which is referred to as a “plaintext”. To ensure secure transmission, the plaintext is encrypted using the receiver’s public key via the RSA encryption module. The output of this process is ciphertext. The communication channel is used to transmit the ciphertext. This channel is supposed to be insecure, but due to the features of RSA encryption, the message remains secure. The ciphertext cannot be decrypted without a private key. To receive the ciphertext, the receiver uses the RSA decipher module to convert the ciphertext back into readable plaintext. The private key, which is kept confidential, is used during the decryption process in which it is applied to decrypt and recover the message.

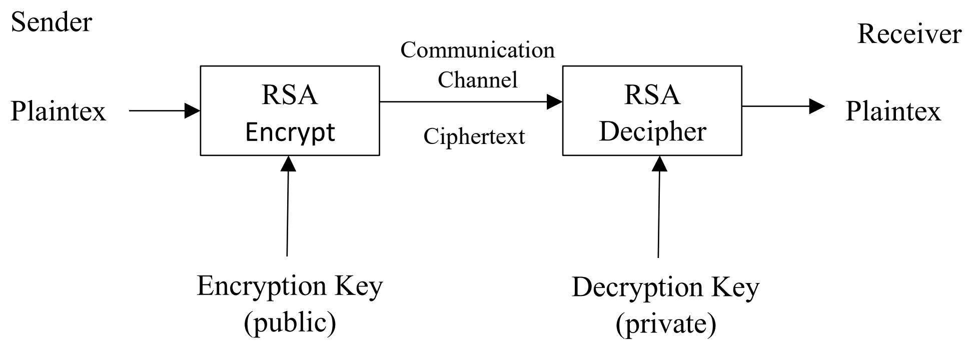


Figure 1: RSA process model

3.1. General steps of RSA

The RSA process involves three key steps: (Educative.io, 2025)

1. Key Generation: In this step, the public, (e, n), and private key, (e, n), are created. Two large prime numbers, p and q are generated. Then, we compute $n = p \times q$ where n is used as the modulus for both the public and private keys. Then, we calculate Euler's totient function:

$$\phi(n) = (p - 1)(q - 1).$$

Next, we choose a public exponent e, where $1 < e < \phi(n)$ and e must be co-prime to $\phi(n)$ (i.e., $gcd(e, \phi(n)) = 1$). Then, we compute the private exponent $d \equiv e^{-1} mod\ \phi(n)$. In other words, d is the modular multiplicative inverse of $e\ modulo\ \phi(n)$.

2. Encryption: Given a plaintext message M (as an integer), using the public key (e, n), we compute the ciphertext C as:

$$C = M^e\ mod\ n$$

3. Decryption: To recover the original message M from the ciphertext C, we use the private key (d, n):

$$M = C^d\ mod\ n$$

Since RSA is considered an Asymmetric key procedure, its security relies partially on the fact that it's easy to choose two random prime numbers, but it's very hard to discover what they are when just given the product of them.

Therefore, typical key sizes are 1,024 or 2,048, or 4,096 bits, unlike Symmetric keys, which are much smaller. The reason RSA is large is that's because there are only so many prime numbers of that size and below. The RSA scheme can only use pairs of prime numbers, whereas the symmetric schemes can use any number of the same size (Educative.io, 2025). But for the sake of simplicity and practicality, we used a small key length in the code.

3.2. Black Box Design

To implement a secure RSA cryptographic system in hardware, both the encryption and decryption algorithms need to be translated into a form that can be executed on digital circuits. This is achieved using Verilog HDL (Hardware Description Language). Figure 2 illustrates the basic I/O of our design, which utilizes the two control signals "go & done" and a reset signal to abort operations, and of course, a clock.

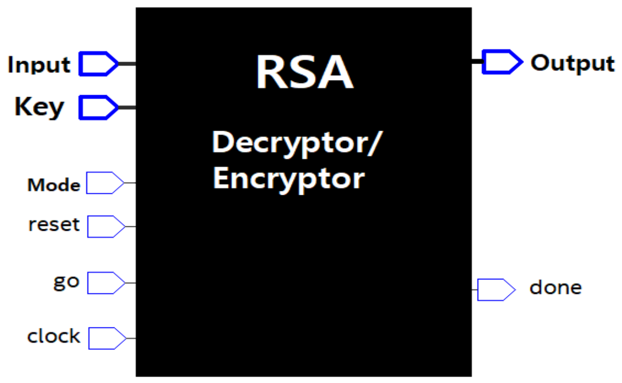


Figure 2: General Black Box Design

The main I/O here are the mode, key, Input, and Output.

Mode: will control which state the digital circuit should be in, either Encryption state or Decryption state.

Key: is the key pair, it depends on whether the Encryption/Decryption mode is chosen.

Input: is going to be either Plaintext or ciphertext, depending on whether Encryption / Decryption mode is chosen.

Output: is going to be either Ciphertext or Plaintext, it depends on whether Encryption / Decryption mode is chosen.

In Figure 3 RSA encryptor takes the encryption key, a mode selection, reset, a 'go' signal to initiate encryption, and a clock signal for synchronization as inputs for plaintext. The encryptor produces ciphertext as its main output and a 'done' signal to indicate completion of the encryption process.

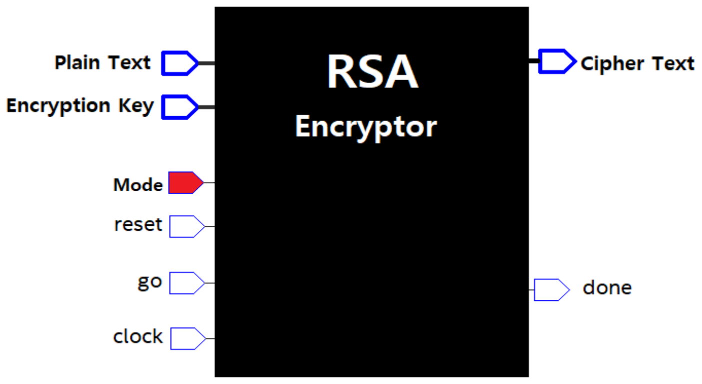


Figure 3: Black Box Design in Encrytion state

Figure 4 depicts the RSA decryptor module, which accepts an encrypted message as input (Ciphertext), private key (Decryption Key), control signals (Mode, reset, go, clock) as inputs. The decryptor produces an unencrypted message (Plaintext) as output and indicates done as completion.

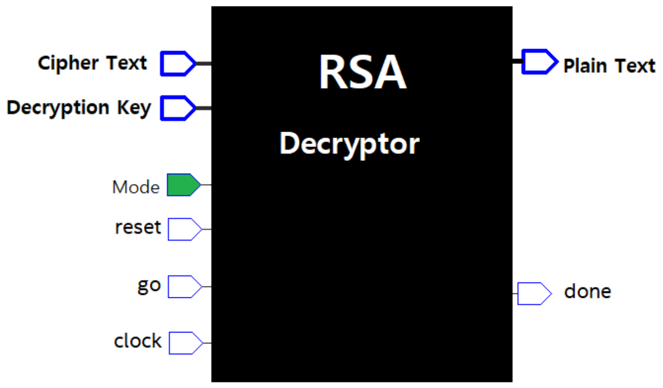


Figure 4: Black Box Design in Decrytion state

3.3. Control Unit Implementation using One-Address ROM

Table 1 presents our control unit implemented using a One-Address Read-Only Memory (ROM). The table explains the control logic based on the current state and the input test conditions by identifying the next state and the control signals.

STATE	AB	TEST	NST	Load	Runinng	Done
S_0	00	00	00	0	0	0
S_1	01	10	10	1	0	0
S_2	10	01	10	0	1	0
S_3	11	11	00	0	0	1

Table 1: Control Unit One Address ROM table

In the control unit, the current state "AB" and the input test condition "TEST" both produce the address to the ROM. The contents of the ROM store the next state "NST" and all the values for the control signals Load, Running, and Done. For example, if the current state S_1 is '01' and the test input is '10', then the ROM is addressed with '0110'. The next state S_2 would be '10' followed by the control signal values '1', '0', and '0'."

3.4. RSA Encryption Decryption System

The RSA encryption and decryption system is an important component of secure digital interactions due to its mathematical foundation and good track record. In our system, we first establish Universal Asynchronous Receiver-Transmitter (UART) communication with the FPGA for the implementation of the RSA algorithm, as shown in Figure 5. The UART provides a serial communication protocol that helps us to send commands to the FPGA and receive the results of the RSA operations.

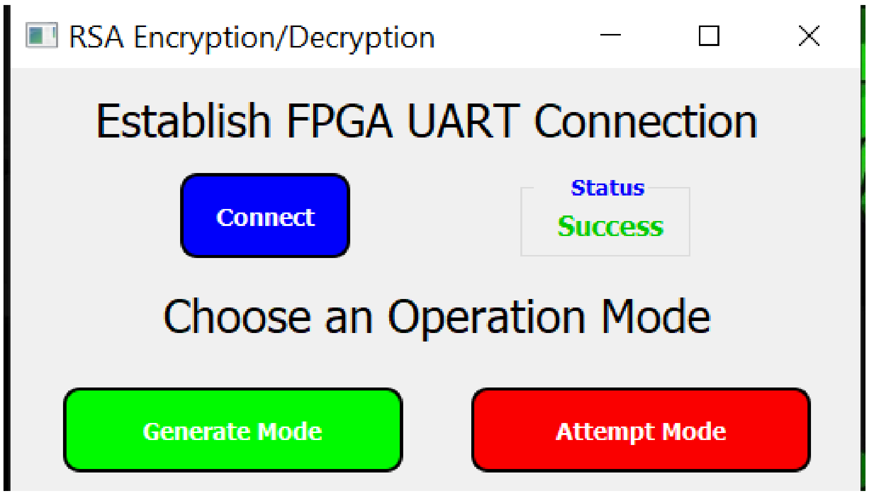
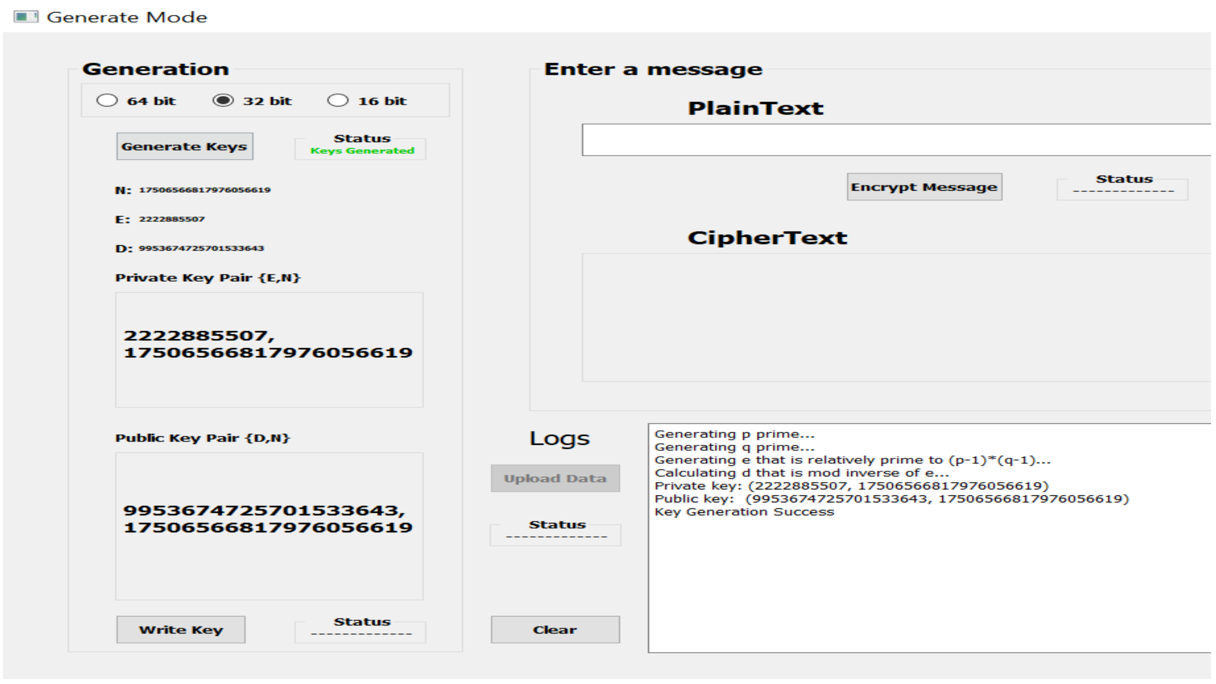


Figure 5: UART communications with the RSA FPGA

After establishing the UART communications, the mode buttons get activated. The "Generate Mode" is where the keys are generated and the plaintext is encrypted with the private key, and the public key is printed onto an RFID key. We choose the prime numbers bit size from the radio buttons in the generate mode window and click the "Generate keys" button to create the key pairs needed, as shown in Figure 6.

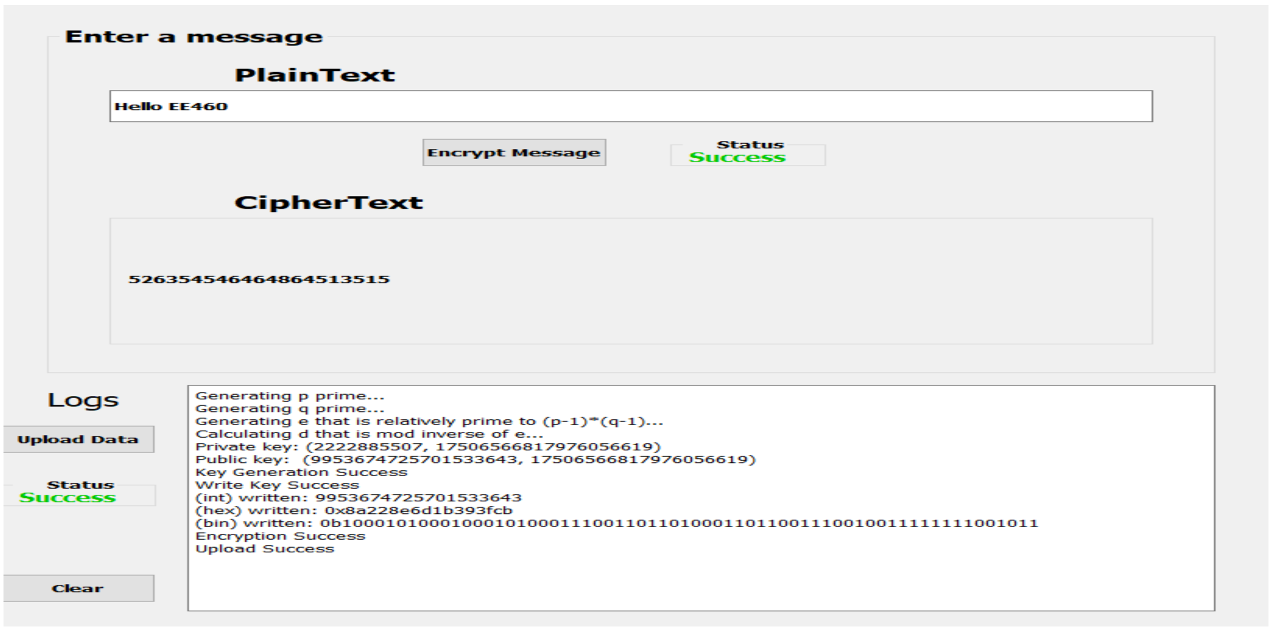


N: 17506566817976056619
E: 2222885507
D: 9953674725701533643
Private Key Pair {E,N}
2222885507,
17506566817976056619
Public Key Pair {D,N}
9953674725701533643,
17506566817976056619

```
Generating p prime...
Generating q prime...
Generating e that is relatively prime to (p-1)*(q-1)...
Calculating d that is mod inverse of e...
Private key: (2222885507, 17506566817976056619)
Public key:  (9953674725701533643, 17506566817976056619)
Key Generation Success
```

Figure 6: Generate window

We can enter our desired plaintext and press “Encrypt,” which will transfer its integer value over to the FPGA to be encrypted. Then we can press the “Upload Data” Button to post our ciphertext on the cloud as in Figure 7.



Hello EE460

526354546464864513515

```
Generating p prime...
Generating q prime...
Generating e that is relatively prime to (p-1)*(q-1)...
Calculating d that is mod inverse of e...
Private key: (2222885507, 17506566817976056619)
Public key:  (9953674725701533643, 17506566817976056619)
Key Generation Success
Write Key Success
(int) written: 9953674725701533643
(hex) written: 0x8a228e6d1b393fcb
(bin) written: 0b1000101000100010100011100110110100011011001110010011111111001011
Encryption Success
Upload Success
```

Figure 7: Encrypt message window

In the attempt mode, the user can go over to another computer and do the same steps for establishing UART communication with the FPGA. Pressing “Fetch” in the attempt window will fetch the ciphertext from the cloud and display it as shown in Figure 8.

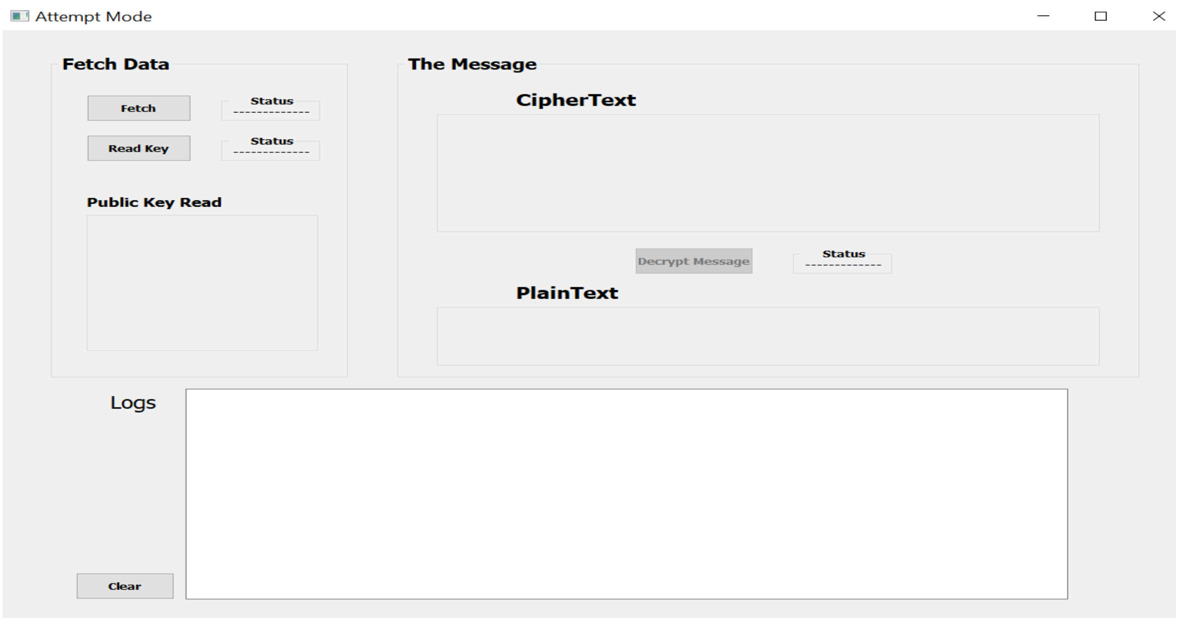
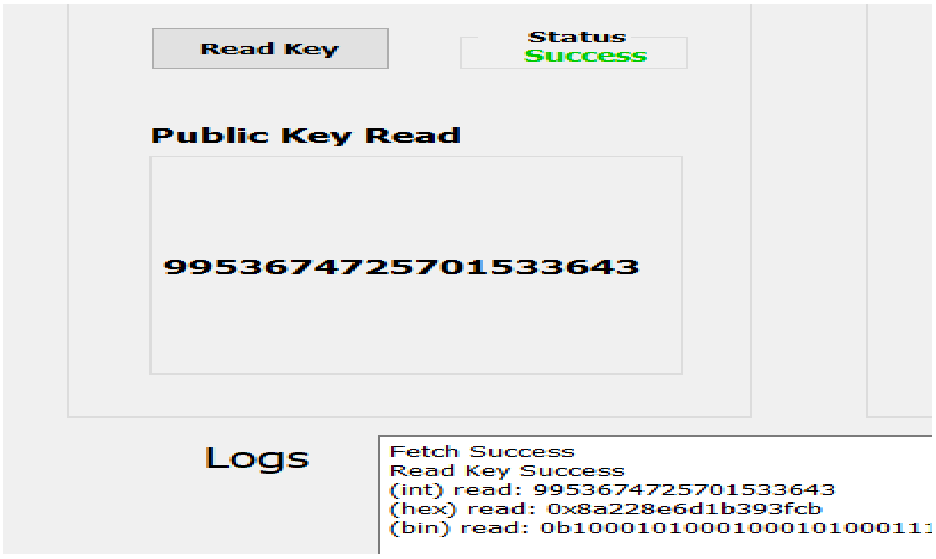


Figure 8: Attempt window

Then pressing the "Read Key" button when placing an RFID Tag over the Reader decodes the public key on it, as shown in Figure 9.

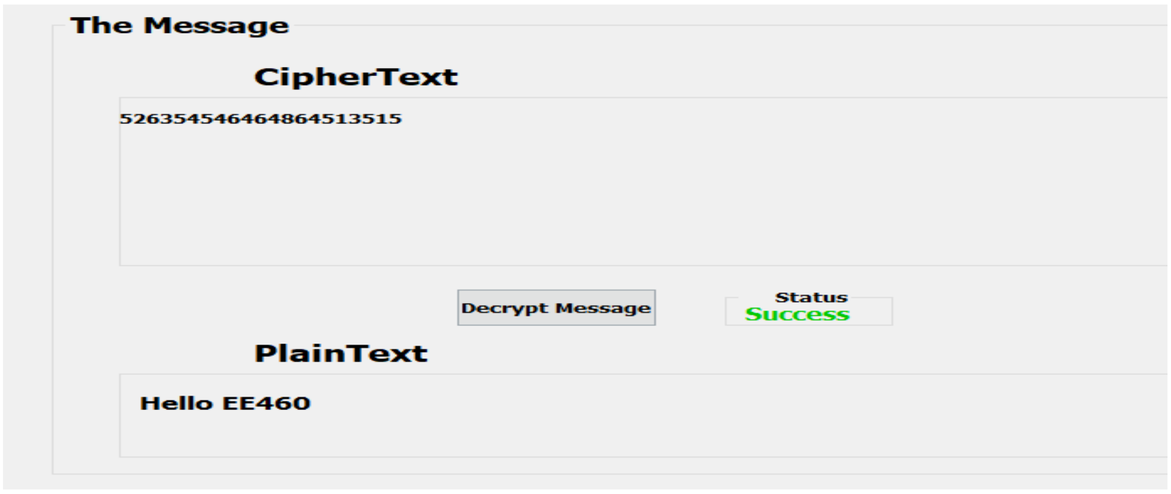


995367472570153364

```
Fetch Success
Read Key Success
(int) read: 995367472570153364
(hex) read: 0x8a228e6d1b393fcb
(bin) read: 0b1000101000100010100011
```

Figure 9: Public key read

Finally, in Figure 10, with the public key at hand and the ciphertext, we can decrypt our message by sending the data over to the FPGA and receiving back the integer, which will be converted to a readable string plaintext.



526354546464864513515

Hello EE460

Figure 10: Decrypt message

4. Conclusion and Recommendations:

In this work, we have implemented the RSA algorithm on an FPGA using Verilog HDL. Our research focus was to develop a hardware-based system that can efficiently handle the computational requirements of RSA, mainly modular multiplication and exponentiation. The result shows that FPGA-based implementations have an advantage over software-based approaches for real-time and embedded cryptographic applications.

Recommendations:

- The research community must focus on optimizing cryptographic hardware for power efficiency and scalability, especially for use in IoT and edge devices.
- Researchers are encouraged to explore the integration of RSA hardware modules with other security protocols for comprehensive system-on-chip (SoC) solutions.
- Future research can investigate implementing RSA with larger key sizes or combining it with other cryptographic algorithms such as AES to enhance performance and security in hybrid systems.
- In digital system design, educational institutions and training programs should incorporate practical FPGA-based cryptographic projects to bridge the gap between theory and application.

The research community can contribute to more efficient, secure, and practical cryptographic systems by continuing to refine RSA hardware implementations and exploring new architectural strategies.

5. References

Abid, H., & Khan, M. F. (2019). 1024-bit RSA encryption algorithm design using Verilog and FPGA implementation. International Journal of Engineering and Advanced Technology (IJEAT), 8(5).

Alshahrani, A. (2017). Implementation of RSA in Verilog. California State University, ScholarWorks Projects.

Arun, T. P., & Dharani, V. (2020). Verilog HDL implementation for an RSA cryptography using shift-sub modular multiplication. Journal of Intelligent and Fuzzy Systems, 17(3).

Educative.io. (2025). What is the RSA algorithm. Retrieved April 10, 2025, from https://www.educative.io/answers/what-is-the-rsa-algorithm

Feng, Y., Nitaj, A., & Pan, Y. (2024). Partial prime factor exposure attacks on some RSA variants. Theoretical Computer Science, 999, 114549.

Kochte, M. A., et al. (2016). Comparison of high concurrency and sequential architectures for RSA on FPGAs. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE).

Kumar, M., & Sharma, S. (2015). Efficient RSA implementation using block RAMs and Montgomery multiplication. International Journal of Computer Applications, 111(4).

Raut, R., & Raut, S. (2006). VLSI implementation of RSA encryption system using ancient Indian Vedic mathematics. arXiv preprint arXiv:cs/0609028.

Saini, C. S. (2017). Design and FPGA implementation of 64-bit RSA cryptosystem. International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering, 5(4).

Ugbedeojo, M., Adebiyi, M. O., Aroba, O. J., & Adebiyi, A. A. (2024). RSA and elliptic curve encryption system: A systematic literature review. International Journal of Information Security and Privacy, 18, 1–27.

Vidhyalakshmi, S., & Ramesh, N. (2013). Cryptosystem: An implementation of RSA using Verilog. International Journal of Computer Applications, 64(17).

Zhang, L., & Zhang, H. (2012). RSA encryption algorithm design and verification based on Verilog HDL. In Proceedings of the 2nd International Conference on Computer Science and Electronic Technology.

Zheng, M. (2024). Revisiting small private key attacks on common prime RSA. IEEE Access, 12, 5203–5211.

Doi: doi.org/10.52132/Ajrsp.e.2025.74.2