VIETNAM JAPAN UNIVERSITY
VIETNAM NATIONAL UNIVERSITY, HANOI

HDL Design for RSA Asymmetric Cryptography

Students : Nguyen Duc Hieu
: Le Minh Kiet

Class : BCSE2022

Supervisors : PhD. Tran Thi Diem
: PhD. Nguyen Van Tinh

Hanoi, June 6, 2025

Abstract

This report details the design, implementation, and simulation-based verification of a modular RSA cryptographic system using the Verilog Hardware Description Language (HDL). Confronting the performance limitations of software-based RSA, this project focuses on creating a hardware foundation for accelerated cryptographic processing. The architecture is composed of several distinct Verilog modules designed to perform the core functionalities of the RSA algorithm. Key components include a key generation module (inverter .v), which implements the Extended Euclidean Algorithm to derive the public exponent (e) and private exponent (d) from two prime inputs. For the encryption and decryption processes, a dedicated modular exponentiation module (`mod_exp.v`) realizes the computationally intensive $M^E \pmod N$ operation by applying the efficient right-to-left binary exponentiation method. The system is orchestrated by a top-level control module (control. v) and supported by underlying arithmetic units. The functional correctness of each module and their integrated operation is validated through a comprehensive simulation strategy using Verilog testbenches, culminating in a full-system test with 128-bit primes for a 256-bit RSA implementation. This work successfully establishes a verified HDL foundation for a hardware-accelerated RSA engine, providing a robust design ready for future work involving FPGA synthesis and on-chip deployment.

Contents

1 Introduction

The Rivest-Shamir-Adleman (RSA) algorithm, a foundational element of modern public-key cryptography [1], plays a critical role in securing digital communications and safeguarding sensitive data. Despite its widespread adoption, the computational demands of RSA, particularly the modular exponentiation of large integers, can impose significant performance limitations on software-based systems. This is especially pertinent in scenarios requiring high throughput or operation within resource-constrained environments. Consequently, the exploration of hardware-accelerated solutions using Hardware Description Languages (HDLs) has garnered considerable attention, as dedicated hardware offers substantial performance gains and a more physically secure platform for cryptographic processing [2]. This project addresses the design, implementation, and verification of core RSA components using the Verilog HDL.

The primary focus is the comprehensive Verilog design and rigorous simulation-based functional verification of a modular RSA system. A notable feature is its parameterization; while tested with small values, the main integration test described in `tb_main.v` uses 128-bit primes, defining a 256-bit RSA system. The goal is to achieve complete functional correctness of all modules, as validated through exhaustive simulation.

2 Theoretical Background and Module Design

2.1 RSA Algorithm Overview

The RSA algorithm involves three primary processes: key generation, encryption, and decryption. The security of the algorithm is based on the mathematical difficulty of factoring the product of two large prime numbers. The overall process is illustrated in the flowchart in Fig. 1.

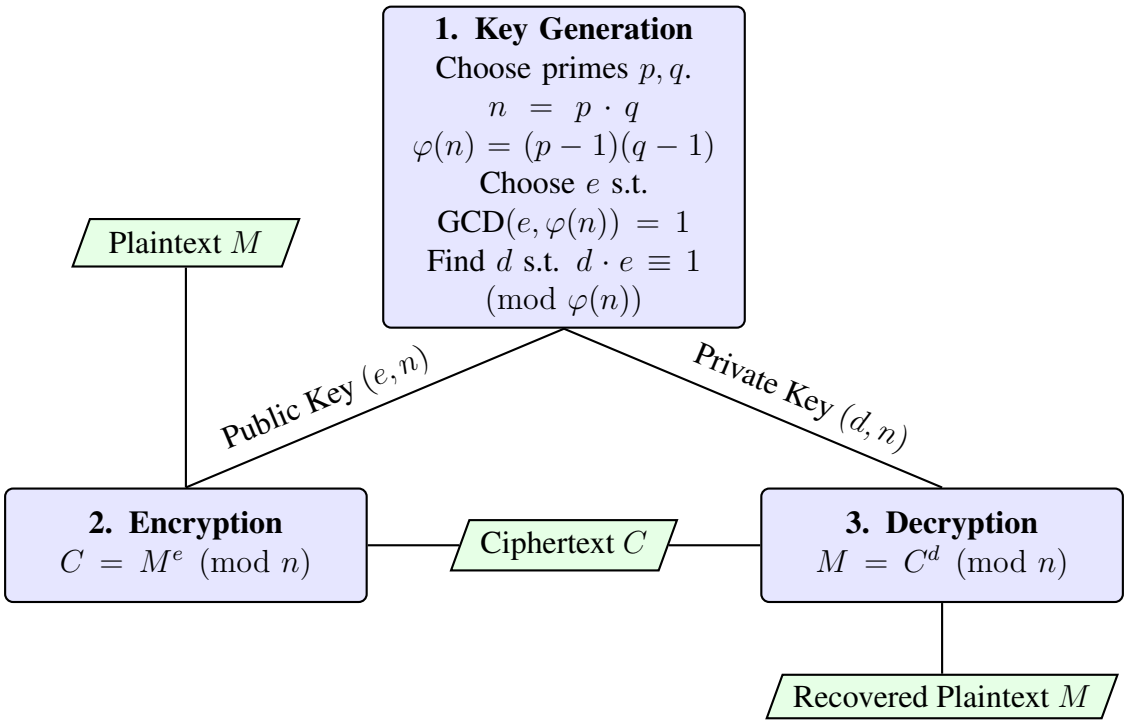

Fig. 1. Flowchart of the complete RSA algorithm, showing the key generation, encryption, and decryption stages and the flow of data between them.

2.2 Module Descriptions

The system is composed of several core Verilog modules, orchestrated by a top-level control unit. The system's functionality is partitioned into the following specific hardware modules.

Control Module (`control.v`): This is the top-level module that integrates all other components and manages the overall operation. Its primary responsibilities include routing the initial prime numbers (p, q) to both the Key Generation module and an internal multiplier to compute the modulus n. It also receives the generated keys (e, d) and selects the appropriate one for the current operation using a multiplexer controlled by the `encrypt_decrypt` signal. Furthermore, it latches the message, modulus, and selected exponent into registers to ensure stable inputs for the main computational engine, and is responsible for issuing reset signals and distributing the system clock to synchronize all sequential sub-modules.

Key Generation Module (`inverter.v`): This sequential module implements a search-based approach to find a valid RSA key pair (e, d). It operates as a finite state machine (FSM) with three primary states: 'UPDATING', 'CHECK', and 'HOLDING'. Upon reset, it initializes a candidate for the public exponent e (starting with 3) and enters the 'UPDATING' state. In this state, the module iteratively performs the steps of the Extended Euclidean Algorithm to compute the Greatest Common Divisor (GCD) of the current e and the totient $\varphi(n)$. Once this calculation is complete, it transitions to the 'CHECK' state. Here, it verifies if the GCD is 1 and if the corresponding private key d is positive. If these conditions are met, a valid pair has been found, and the FSM moves to the 'HOLDING' state, asserting the 'finish' signal. If not, the module discards the current e, increments it to the next odd number, and returns to the 'UPDATING' state to repeat the process with the new candidate.

Modular Exponentiation Module (`mod_exp.v`): As the computational core of the system, this module performs the modular exponentiation required for both encryption and decryption. It implements the right-to-left binary method, also known as the square-and-multiply algorithm, as a simple finite state machine (FSM) with two states: 'UPDATE' and 'HOLD'. Upon reset, the module latches the base, exponent, and modulus into internal registers, initializes the result to 1, and enters the 'UPDATE' state. In each clock cycle within this state, it checks the least significant bit (LSB) of the exponent. If the LSB is 1, it multiplies the current result by the base and takes the modulus (the "multiply" step). Concurrently, it squares the base and takes the modulus (the "square" step), and right-shifts the exponent by one bit. This process repeats until the exponent becomes zero, at which point the FSM transitions to the 'HOLD' state, indicating the calculation is complete and the final result is valid.

Support Modules (`mod.v`, `divider.v`, etc.): This category comprises a set of purely combinational arithmetic modules that perform basic operations like division and modulo. These are utilized as foundational building blocks by the main sequential modules.

3 Testing and Verification Strategy

Functional correctness is verified through a hierarchical simulation strategy, starting with unit tests for each module and culminating in a full-system integration test.

3.1 Unit Testing

Each module is tested individually using a dedicated Verilog testbench:

- **Modulo Module (`mod.v`)**: Verified with `mod_tb.v`. A key test case sets `a=12345` and `n=23`, expecting quotient `q=536` and remainder `r=17`.
- **Key Generation Module (`inverter.v`)**: Verified with `tb_inverter.v`. For inputs `p=7` and `q=5`, the test expects the final outputs `e=5` and `d=5`.
- **Modular Exponentiation Module (`mod_exp.v`)**: Verified with `tb_mod_exp.v`. The test calculates $5^{567} \pmod{13}$, expecting the result `8`.

3.2 Integration Testing

The final verification is performed by the main testbench (`tb_main.v`), which tests the entire control module in a realistic 256-bit decryption scenario. It provides 128-bit primes and a 256-bit ciphertext to confirm end-to-end functionality.

4 Simulation Waveforms

This section presents the simulation results for each testbench, which are essential for visually confirming the correct temporal behavior of the hardware modules.

4.1 Modulo Operation

The modulo operation module (`mod.v`) is a fundamental arithmetic component in the RSA system. It is designed to compute both the quotient and the remainder of a division operation. Unlike most other modules in this design, it implements purely combinational logic, meaning its outputs are a direct function of its inputs without requiring clock cycles.

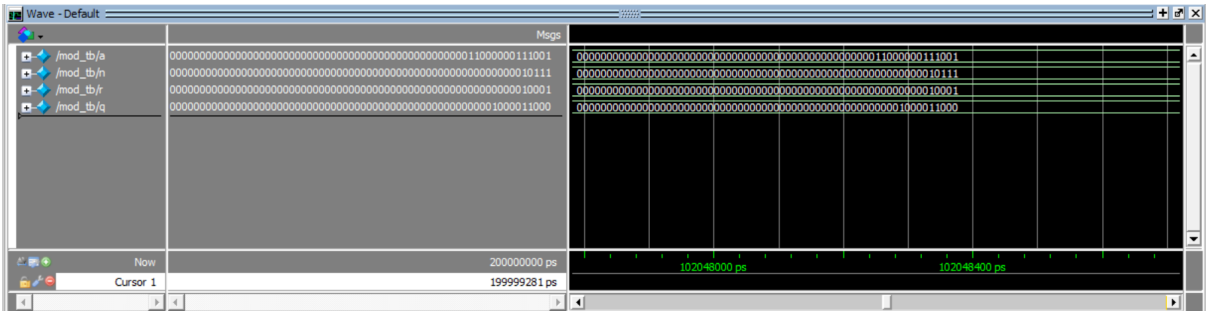

Fig. 2. Simulation waveform for the combinational modulo operation module (`mod.v`). The test verifies the operation for inputs `a = 12345` and `n = 23`, confirming the outputs stabilize at the expected values: quotient `q = 536` and remainder `r = 17`.

Waveform Analysis: The provided waveform confirms the correct and instantaneous behavior of the module.

Direct Functional Verification The testbench applies the input values `a = 12345` (hex: `3039`) and `n = 23` (hex: `0017`). As depicted in the waveform, the module's outputs immediately stabilize to the correct, mathematically calculated values. The quotient output, `q`, correctly shows `536` (hex: `0218`), and the remainder output, `r`, correctly shows `17` (hex: `0011`). There is no clock signal involved, and the absence of state changes over time underscores the module's combinational nature.

System Role While simple, the correct operation of this module is critical. It serves as a foundational building block that is heavily utilized by the more complex, iterative modules within the system, such as the modular exponentiation engine which performs repeated division and modulo calculations. The successful verification of this unit test therefore provides confidence in the arithmetic primitives supporting the entire cryptographic core.

4.2 Key Generation

The key generation module (`inverter.v`) is responsible for calculating the public and private exponents (e and d) from two initial prime numbers, p and q. This is achieved by implementing the Extended Euclidean Algorithm. The waveform in Fig. 3 shows a test case with small prime inputs to verify the algorithmic logic.

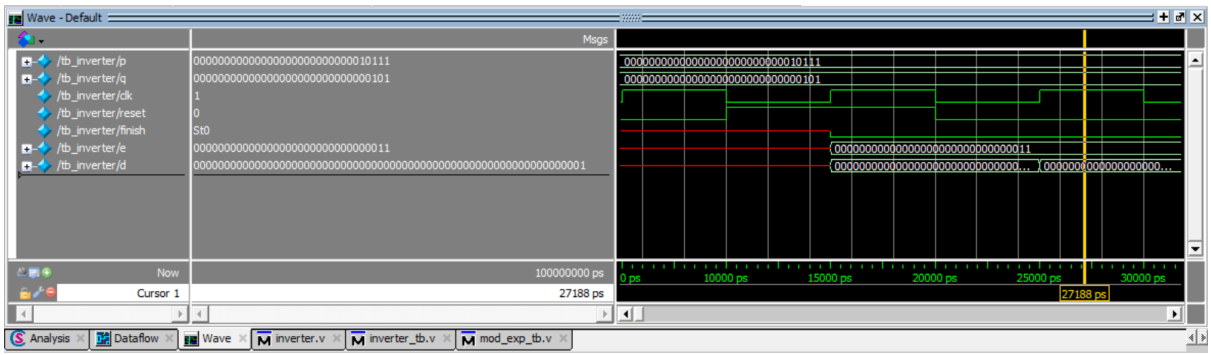

Fig. 3. Simulation waveform for the key generation module (`inverter.v`) with inputs `p=7` and `q=5`. This snapshot at `t = 27188 ps` shows intermediate values during the iterative calculation. The `finish` signal is low, indicating the algorithm has not yet completed.

Waveform Analysis: The waveform provides a snapshot of the module's state during its execution, confirming the iterative nature of the key-finding process.

Setup and Intermediate State The module is initialized with prime inputs `p = 7` and `q = 5`. From these, it first calculates the totient $\varphi(n) = (p-1)(q-1) = 6 \times 4 = 24$. The core task is then to find an exponent pair (e, d) such that $e \cdot d \equiv 1 \pmod{24}$. The snapshot at `t = 27188 ps` captures the module mid-calculation. At this moment, the `finish` signal is low, and the output buses show intermediate values (e.g., `e = 11`, `d = ...`), which are temporary results generated during the steps of the Extended Euclidean Algorithm. These are not the final keys.

Expected Completion If the simulation is allowed to run to completion, the module will continue its search. For $\varphi(n) = 24$, a valid public exponent is `e=5`, since $\text{GCD}(5, 24) = 1$. The corresponding private exponent would be `d=5`, because $5 \times 5 = 25 \equiv 1 \pmod{24}$. Therefore, upon completion, the `finish` signal would be asserted high, and the output buses would stabilize at these final, correct values: `e=5` and `d=5`. This snapshot, while not showing the final result, is crucial for verifying the module's sequential logic is active and progressing as designed.

4.3 Modular Exponentiation

The modular exponentiation module (`mod_exp.v`) implements the right-to-left binary exponentiation algorithm. The test case shown in Fig. 4 verifies the calculation of $5^{567} \pmod{13}$.

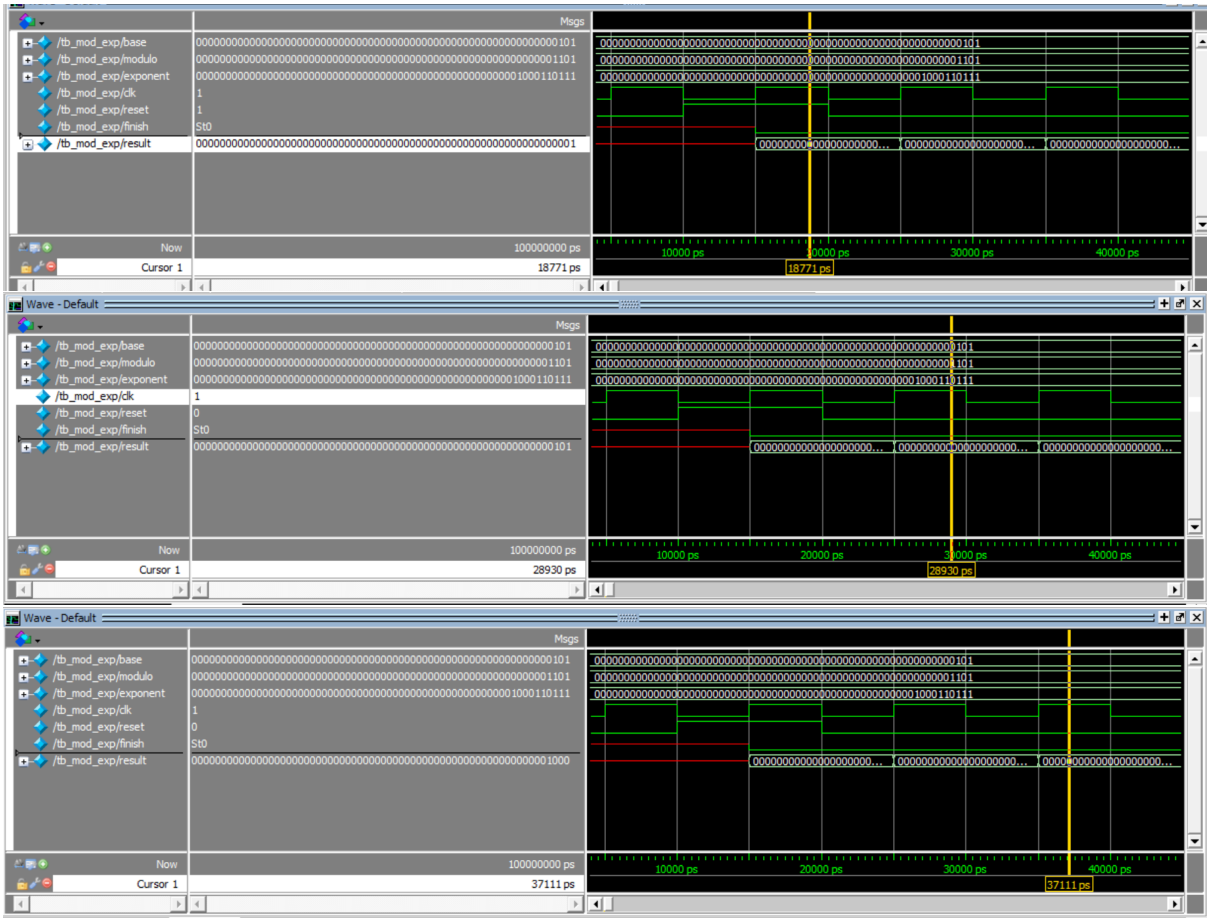

Fig. 4. A sequence of simulation snapshots for the modular exponentiation module (`mod_exp.v`) calculating $5^{567} \pmod{13}$. The top waveform shows an early stage, the middle one an intermediate calculation, and the bottom one the final result where `finish` is asserted and `result` stabilizes at the correct value of 8.

Waveform Analysis: The behavior observed confirms the correct step-by-step execution of the algorithm.

Initialization After the initial `reset` pulse, the module loads the inputs: `base` = 5 (`'h05'`), `exponent` = 567 (`'h237'`), and `module` = 13 (`'h0D'`). The internal `result` register is initialized to 1, and the `finish` signal is de-asserted (low).

Iterative Calculation The module iterates based on the bits of the exponent (567 is `1000 110111` in binary). The progression is as follows. The snapshot at `18771 ps` shows the `result` bus holding the initial value `1`. Subsequently, the snapshot at `28930 ps` displays an intermediate value. Here, the `result` bus shows `5`, corresponding to the first algorithmic step where the LSB of the exponent is 1:

$$result = (1 \times 5) \pmod{13} = 5$$

The module then continues this process of conditional multiplication and squaring for each bit of the exponent, updating the `result` register accordingly.

Completion The final snapshot (at `37111 ps`) shows the final state of the operation. The `result` bus now holds the final, correct value of `8` (`'h08'`). Critically, the `finish` signal is asserted high (`1`), indicating that the computation is complete and the output value on the `result` bus is valid.

This sequence of states provides strong evidence that the modular exponentiation module correctly implements the iterative logic of the binary modular exponentiation algorithm.

4.4 Full System Integration

The final verification stage involves the main testbench (`tb_main.v`), which simulates the entire RSA system's end-to-end functionality. The waveforms in Fig. 5 illustrate a complete encryption and subsequent decryption cycle, controlled by the `encrypt_decrypt` signal.

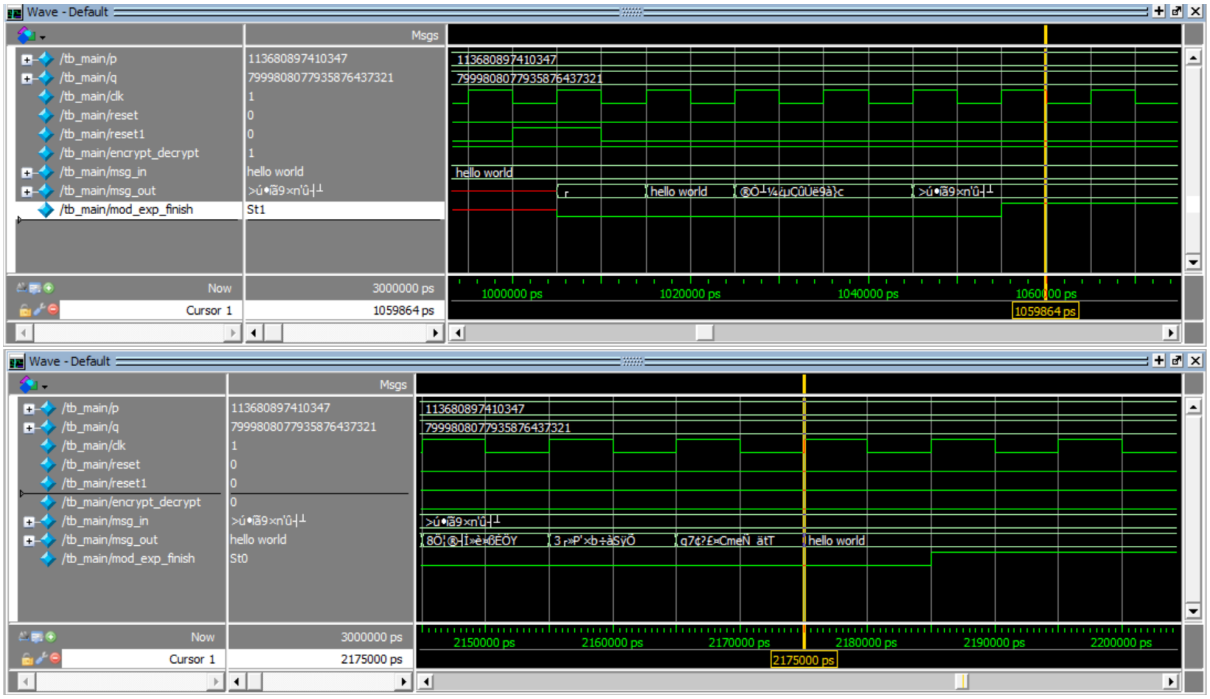

Fig. 5. Top-level simulation showing the complete RSA encryption-decryption cycle from the main testbench (`tb_main.v`). The top image shows the encryption phase (`encrypt_decrypt=1`), while the bottom image shows the decryption phase (`encrypt_decrypt=0`), successfully recovering the original message.

Waveform Analysis: The two snapshots demonstrate a successful round-trip test of the cryptographic core.

Encryption Phase (`encrypt_decrypt = 1`) The top waveform displays the encryption process, initiated by the first `reset` pulse. The `encrypt_decrypt` signal is set to `1`. The plaintext message, "hello world", is provided on the `msg_in` bus. The module processes this input, and upon completion (indicated by `mod_exp_finish` going high), the resulting ciphertext appears on the `msg_out` bus. This non-readable output is the correct, encrypted form of the original message.

Decryption Phase (`encrypt_decrypt = 0`) The bottom waveform shows the subsequent decryption process, initiated by a delayed `reset1` pulse. The `encrypt_decrypt` signal is now set to `0`. Crucially, the ciphertext generated in the previous phase is fed back as the input on the `msg_in` bus. After the core performs the decryption calculation, the `msg_out` bus correctly displays the original "hello world" message. The assertion of the `mod_exp_finish` signal confirms the completion of this phase.

This successful recovery of the original plaintext after a full encryption-decryption cycle provides definitive verification of the entire system's functional correctness.

5 Conclusion

This report has successfully detailed the completion of the HDL design and unit-level verification for the core components of a modular, 256-bit RSA cryptographic system. A comprehensive verification framework is now in place, culminating in the `tb_main.v` testbench for full system integration testing. The immediate next step is the execution of these simulations to capture and analyze waveforms against the expected behavior detailed herein. Looking forward, future work will extend beyond this scope to involve FPGA synthesis for analyzing resource utilization and timing performance, followed by physical hardware implementation and on-chip verification.

6 References

References

[1] Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press (1996)

[2] Tahir, A.S.: Design and Implementation of RSA Algorithm using FPGA. International Journal of Computers & Technology. 14(12), 6433–6442 (2015)