Digital Circuit Lab 2 Report

林軒霆, 蔡丞彥, 侯奕安

October 14, 2025

1 Introduction

In Lab 2, we implemented an RSA-256 decryption module. The overall structure includes a `core.sv` for the calculation and a `wrapper.sv` for controlling the core and transferring data between RS232 and the FPGA. The core uses exponentiation by squaring to reduce the number of multiplications and applies the Montgomery algorithm to calculate $ab \bmod N$ faster.

2 File Structure

For Lab2, our code is written in the provided `Rsa256Core.sv` and `Rsa256Wrapper.sv` files, and no other code is modified or added. The files we submitted are shown in the figure above. For demonstration, replacing the original files with these would be sufficient.

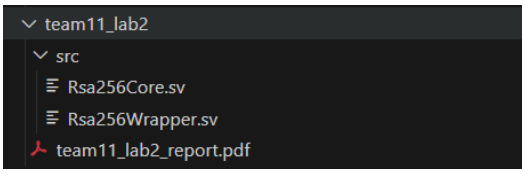


Figure 1: Hierarchy of Files

3 System Architecture

The top module is `Rsa256Wrapper`, which includes a submodule `Rsa256Core`. The core contains two submodules, `RsaPrep` and `RsaMont`. The first submodule prepares $a\,2^{256} \bmod N$ and $b\,2^{256} \bmod N$ for the Montgomery algorithm, while `RsaMont` performs the algorithm itself. The relationship between signals is shown in the figure below (clock and reset signals are synchronized with the wrapper input and are thus omitted).

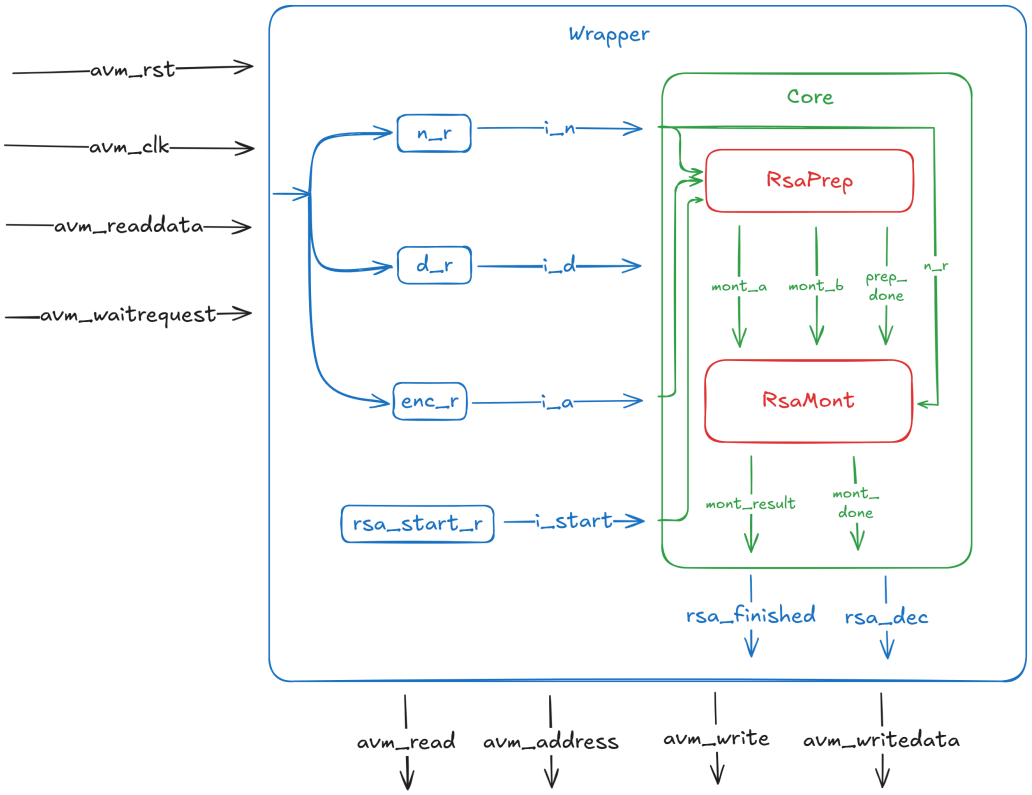


Figure 2: Architecture of Our Design

4 Hardware Scheduling

4.1 FSM in Wrapper

The FSM used in the wrapper is shown in the figure below. We use four states, and the transitions between them are straightforward. The initial state is `S_GET_KEY`:

1. `S_GET_KEY` to `S_GET_DATA`: When the input bus is ready, the input address is `RX_BASE`, and the byte counter has decreased to zero. This means that the key has been successfully received.

2. `S_GET_DATA` to `S_WAIT_CALCULATE`: When the input bus is ready, the input address is `RX_BASE`, and the byte counter has decreased to zero. This means that the encrypted data has been successfully received.

3. `S_WAIT_CALCULATE` to `S_SEND_DATA`: When the RSA decryption is done.

4. `S_SEND_DATA` to `S_GET_KEY`: When the output bus is ready, the output address is `TX_BASE`, and the byte counter has decreased to zero. This means that the decrypted data has been successfully sent.

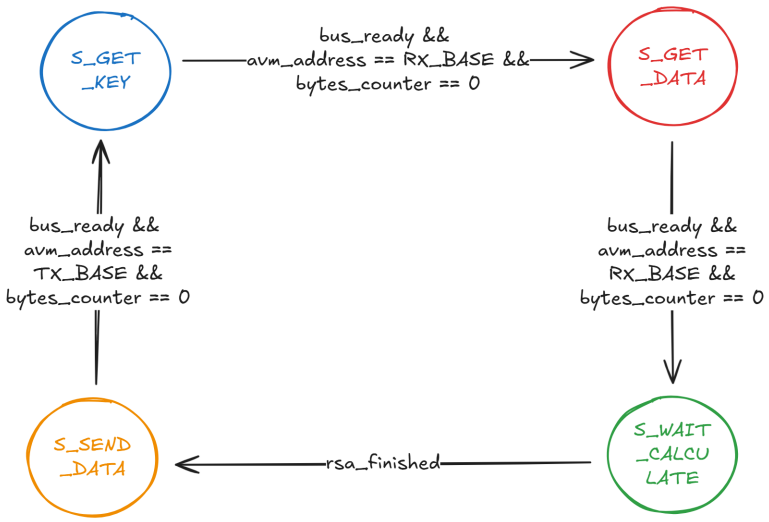


Figure 3: FSM in Wrapper

4.2 FSM in Core

The FSM used in the core is shown in the figure below. The transitions between states are as follows:

1. `S_IDLE` to `S_PREP`: When the start signal is triggered.
2. `S_PREP` to `S_CALC`: When the preparation is done, which is signaled by `RsaPrep`.
3. `S_CALC` to `S_MONT`: When `bit_counter` < 256 (still processing bits).
4. `S_MONT` to `S_CALC`: When the current Montgomery calculation is done and no extra multiplication is needed.
5. `S_MONT` to `S_MONT`: When the current Montgomery calculation is done and one extra multiplication ($t \times t \bmod N$) is needed.
6. `S_CALC` to `S_DONE`: When `bit_counter` ≥ 256 (all bits processed).

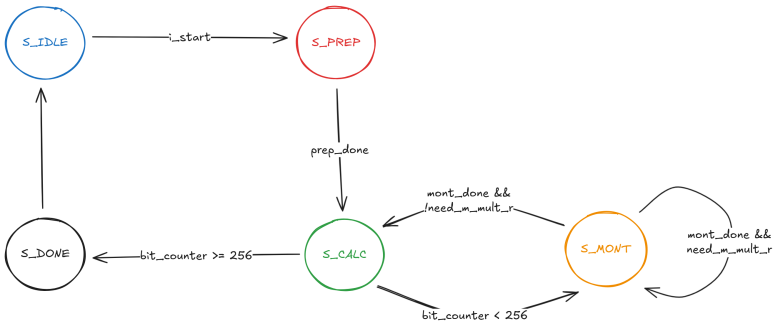


Figure 4: FSM in Core

4.2.1 FSM in RsaPrep and RsaMont

The FSM used in RsaPrep is shown in the figure below. The transitions between states are

1. `S_IDLE` to `S_REDUCE`: When start signal is triggered.

2. `S_REDUCE` to `S_REDUCE`: When `x` >= `i_n`. Make sure `x` < `n` before proceeding to `S_REDUCE`.

3. `S_REDUCE` to `S_DOUBLE`: When `x` < `n` `&&` `counter` < 256. In this state, perform the operation of $2 \times$ `x` mod `i_n`..

4. `S_REDUCE` to `S_FINAL`: When `x` < `n` `&&` `counter` >= 256, now `x` = `y` $\times 2^{256}$ mod `i_n`.

5. `S_FINAL` will only take one cycle and then return to `S_DONE`. Here, the last modulus operation is performed to ensure the output `t` < `i_n`.

6. In `S_DONE`, the finished signal is set to 1, and the state returns to `S_IDLE` to wait for the next use.

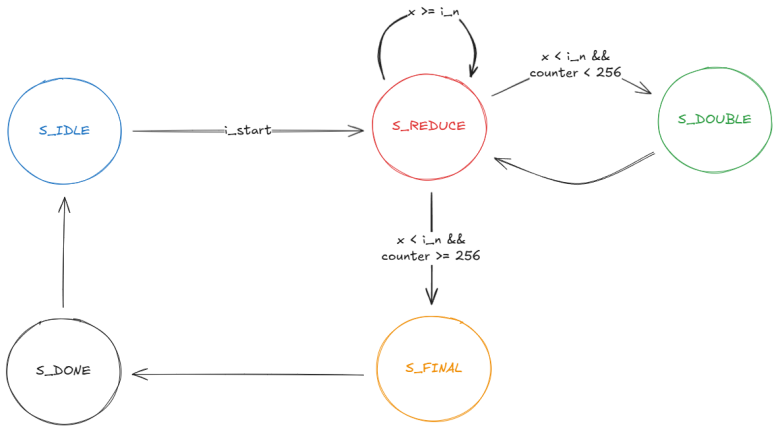


Figure 5: FSM in RsaPrep

The FSM used in RsaMont is shown in the figure below. The transitions between states are

1. `S_IDLE` to `S_CALC`: When start signal is triggered.

2. `S_CALC` to `S_REDUCE`: When `counter` >= 256. `S_CALC` iterates through all 256 bits of operand `a`, performing Montgomery multiplication's core algorithm. For each bit position `i`, it examines `a[i]`:

- `a[i]` = 1: Add `b` to `m`, then divide by 2. Since division must be exact, the algorithm ensures `b` + `m` is even before the right shift. It checks the XOR of the LSBs: if `m[0]` $\oplus$ `b[0]` = 1 is odd, it adds the odd modulus n to make the sum even. Otherwise, `m` = (`m` + `b`) >> 1.
- `a[i]` = 1: Only divide `m` by 2. If `m[0]` = 1, add n first: m = `m` + `n` >> 1. If `m[0]` = 0: m = `m` >> 1.

When all 256 bits are processed, the state transitions to `S_REDUCE`.

3. `S_REDUCE` will only take one cycle and then return to `S_DONE`. Here, the last modulus operation is performed to ensure the output `m` < `n`.
4. In `S_DONE`, the finished signal is set to 1, and the state returns to `S_IDLE` to wait for the next use.

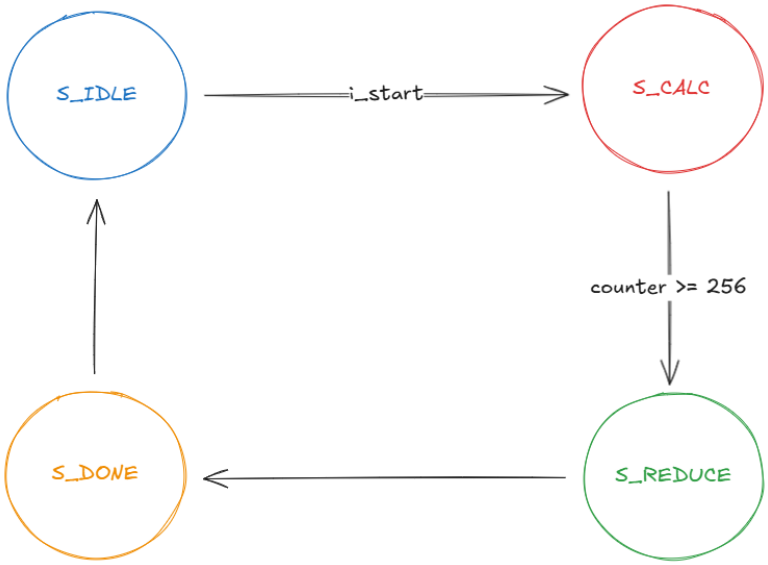


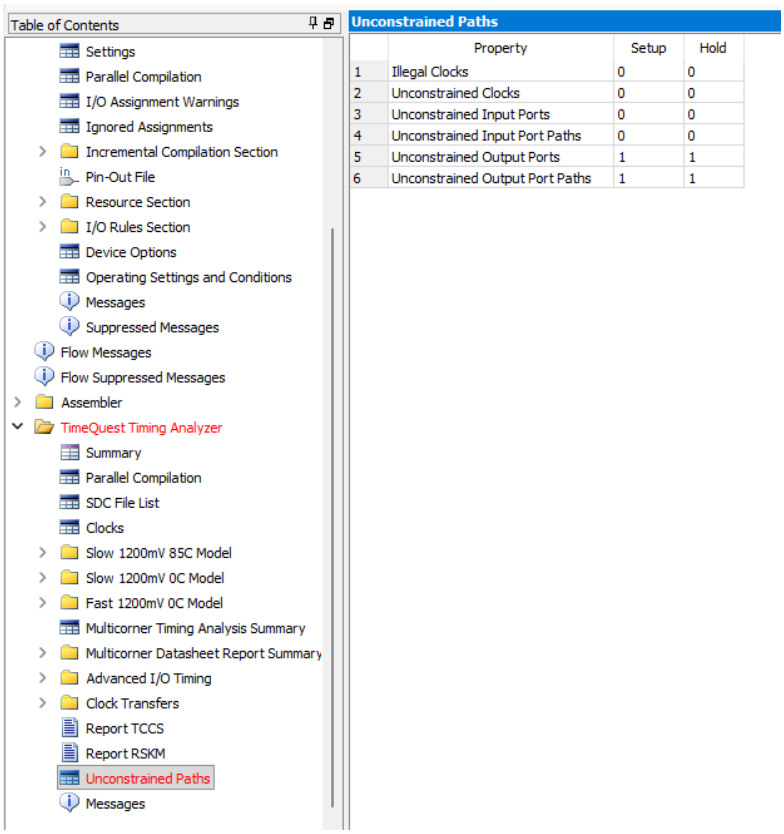
Figure 6: FSM in RsaMont

5 Screen shot of Fitter Summary

Fitter Summary	
Fitter Status	Successful - Tue Oct 14 18:01:33 2025
Quartus II 64-Bit Version	15.0.0 Build 145 04/22/2015 SJ Full Version
Revision Name	DE2_115
Top-level Entity Name	DE2_115
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	8,070 / 114,480 (7 %)
Total combinational functions	6,911 / 114,480 (6 %)
Dedicated logic registers	4,015 / 114,480 (4 %)
Total registers	4015
Total pins	518 / 529 (98 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	1 / 4 (25 %)

Figure 7: Fitter Summary

6 Screen shot of Timing Analyzer



	Property	Setup	Hold
1	Illegal Clocks	0	0
2	Unconstrained Clocks	0	0
3	Unconstrained Input Ports	0	0
4	Unconstrained Input Port Paths	0	0
5	Unconstrained Output Ports	1	1
6	Unconstrained Output Port Paths	1	1

Figure 8: Timing Analyzer

7 Problems Encountered

The most significant obstacle we encountered in this lab was related to Quartus. Since we needed to include a `.qsys` file in this experiment, the overall procedure in Quartus was much more complicated than in Lab 1. During our first attempt, the `.sv` files could not run on the FPGA board. After several hours of debugging, we finally discovered that the issue was caused by not importing the `.qsf` file. This experience taught us that carefully following every step in the tutorial can save a huge amount of time.