NTNU – Trondheim
Norwegian University of
Science and Technology

TFE4141
Design of Digital Systems 1

High Speed RSA-Encryption/Decryption

Cristian Gil Morales (500306)
Felix Allan Schöpe (502139)

November 25, 2018

Abstract

I find VHDL is like swimming with a lifeguard on duty, whereas Verilog is like swimming with a life buoy hanging by the poolside. You decide for yourself.

The present report presents a possible implementation of the RSA algorithm's core, designed with the Hardware Description Language VHDL.

Given an input message (and its key/modulus as well), the system should encrypt/decrypt any input data properly, trying to offer the most reliable, fastest and most compact possible solution.

The academic aim of this project is to learn the hardware designing approach and to work with the basic functions of VHDL.

Contents

List of Figures

List of Tables

1 Introduction

The necessity of designing reliable more complex hardware is increasing exponentially along the years, so the industry has evolved to face such a problem: new approach for designing digital hardware has been invented and its name is Electronic Design Automation (EDA). EDA is a category of software tools for designing electronic systems such as integrated circuits and printed circuit boards [1].
Such an approach lets the designer forget everything about the low level part of the system and focus on the high level point of view. That is to say that the designer is now able to leave physics and electrical connections behind, and just concentrate on the big design, in the idea that he/she is trying to make a new reality. Now Application-Specific Integrated Circuits (ASICs), microprocessors (uP) and Programmable Logic Devices (PLD) are really feasible to create.

An integral part of the EDA is the so called Hardware Description Language (HDL). The HDL enables a precise, formal description of an electronic circuit that allows for the automated analysis and simulation of an electronic circuit. It looks much like a programming language such as C: it is a textual description consisting of expressions, statements and control structures. But the reality is that they are quite different to each other, as the main difference would be that the HDLs explicitly include the notion of time.
In addition, they also allow for the synthesis of a HDL description into a Netlist (a specification of physical electronic components and how they are connected together), which can then be placed and routed to produce the set of masks used to create an integrated circuit.

The most known HDLs are Verilog and VHDL, every one with its strengths and weaknesses. The present report shows a system implementation programmed with VHDL. And out of thousands of possibilities that the VHDL allows to program, for this project has been determined to design a cryptographic algorithm: the RSA cryptosystem.

2 Theoretical Background

2.1 Necessary background

To understand the further sections, the reader should be familiar with the next terms:

- RSA cryptographic algorithm
- Hardware Description Languages: VHDL, Verilog...
- High-level programming languages: Python, C, Java...
- Field Programmable Gate Array
- Software suites for synthesis & analysis of HDL designs: Vivado, Lattice Diamond...
- Finite State Machines
- Timing diagrams for digital systems simulation (testbench)

2.2 Acronyms

Hereafter the following acronyms are used so as to make the document more readable.

Acronym	**Description**
AXI	Advanced Extensible Interface
DMA	Direct Memory Access
EDA	Electronic Design Automation
FF	Flip Flop
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
HW	Hardware
IP	Intellectual Property
LUT	LookUp Table
RSA	Rivest-Shamir-Adleman
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Table 2.1: Acronyms used in this document

3 Problem description and analysis

3.1 Problem presentation

RSA (Rivest-Shamir-Adleman) is one of the first public-key cryptosystems and is widely used for secure data transmission. First publicly described in 1978, such a cryptosystem consists on the fact that the encryption key is public and it is different from the decryption key, which is kept secret (private).

In RSA, this asymmetry is based on the practical difficulty of the factorization of the product of two large prime numbers, the "factoring problem".

A user of RSA creates and then publishes a public key based on two large prime numbers, along with an auxiliary value. The prime numbers must be kept secret. Anyone can use the public key to encrypt a message, but with currently published methods, and if the public key is large enough, only someone with knowledge of the prime numbers can decode the message feasibly [2].

3.2 Main requirements

The requirements for the present project are listed below:

1. The design must implement the RSA encryption algorithm.
2. Encrypt/decrypt a message of 256 bits length as fast possible.
3. The design must fit inside the Zynq-7000 FPGA on the Pynq Z1 board.
4. The RSA design must be integrated as an hardware accelerator inside the Zynq SoC. It must be managed by the CPU and made accessible through the Jupyter notebook interface.
5. The design should implement memory mapped status registers, status registers, performance counters and other mechanisms for debugging of features and performance at system level.
6. The design must have one AXI-Lite Slave interface to enable access of memory-mapped registers. The design must have one AXI stream slave interface for messages that shall be encrypted (decrypted) and one AXI stream master interface for messages that have been encrypted (decrypted).

7. The design could support a configurable message/key size 128, 256.
8. A verification plan must be created and the design must be verified and meet the quality targets set in the verification plan. The verification plan must include the development of unit level testbench(es) as well as testing on the FPGA. The design must work.
9. The verification plan could include collection of code coverage, functional coverage as well as simple assertions.
10. A functional high-level model of the algorithm must be created.
11. A microarchitecture diagram must be created before any VHDL code is written.

3.3 Initial investigation

Before starting working on this project, some background is necessary to understand how the RSA cryptosystem works. Only in this way we will have the big picture of all steps: which ones must be implemented and which ones have already been implemented (if so).

The RSA algorithm involves four steps: key generation, key distribution, encryption and decryption.

1. Key generation
   - Choose two prime numbers p and q (random and similar in magnitude).
   - Compute n = pq (it is the modulus for both the public and private keys).
   - Compute $\lambda(n) = lcm(\lambda(p), \lambda(q)) = lcm(p-1, q-1)$.
   - Choose an integer e such that $1 < e < \lambda(n)$ and $gcd(e, \lambda(n)) = 1$. e and λ(n) are coprime.
   - Determine d as $d \equiv e^{-1} \pmod{\lambda(n)}$. Then d is the modular multiplicative inverse of e modulo $\lambda(n)$.
2. Key distribution
   The public key consists of the modulus n and the public (or encryption) exponent e. This key must have it the sender.
   The private key consists of the private (or decryption) exponent d, which must be kept secret by the receiver. p, q, and $\lambda(n)$ must also be kept secret because they can be used to calculate d.
3. Encryption
   The input message M (unpadded plaintext) into an integer m (the padded plaintext), such that 0?m<n by using an agreed-upon reversible protocol known as a padding scheme. Then it computes the ciphertext c using the public key, corresponding to:
   $$c \equiv m^e \pmod{n} \tag{3.1}$$

4. Decryption
   The received data c can be transformed into m by using the private key exponent d by computing:
   $$m \equiv c^d \pmod{n} \tag{3.2}$$
   Finally, with m stored, the original message M can be recovered by reversing the padding scheme.

Overall, the encryption/decryption process (both are the same as the formula is the same) must be implemented as the core of the RSA system.
In addition for this particular project, the message (M), public/private key (E/D) and modulus (N) are directly given from an external interface. They are chunks of 256 bits each, and already prepared with the mathematical relations they must fulfill each other.

3.4 Discuss/Analyze/Conclude

To implement the requested functionality, this project obligates us to think in a different way than we are used to: the design must be done with a Hardware approach, not a Software approach. For such a thing, mastering the basics of VHDL is mandatory.

4 Design exploration and presentation of solution

4.1 Selected algorithm

There are different designs to implement the RSA cryptographic algorithm, with different performance and complexity. Selecting the right algorithm, the number of used area and the efficiency rate would change dramatically.
Following the design principle KISS (Keep It Simple, Stupid), the chosen algorithms are:

```
1.  $C := 1$ ; $P := M$
2.  **for** $i = 0$ **to** $h - 1$
2a.     **if** $e_i = 1$ **then** $C := C \cdot P (\mathrm{mod}\ n)$
2b.     $P := P \cdot P\ (\mathrm{mod}\ n)$
3.  **return** $C$
```

(a) RL Binary Method

```
1.  $P := 0$
2.  **for** $i = 0$ **to** $k - 1$
2a.     $P := 2P + A \cdot B_{k-1-i}$
2b.     $P := P \bmod n$
3.  **return** $P$
```

(b) Interleaving Multipl and Reduct

Figure 4.1: Algorithms to implement RSA cryptosystem

These two algorithms are:

- **RL Binary Method:** This algorithm is the main one and outputs the final encrypted/decrypted message. Being P(1) the input message M and C=1, both are updated in a for loop as long as the length of the key E. P(1) will be updated in every iteration, but C only when the particular bit of the key E is 1 (E is checked bitwise from right to left).
To update both P(1) and C, the modular operation is required. Unfortunately, this operation is quite time-consuming, so a second algorithm is required to make such an operation much easier (so faster).

- **Interleaving Multiplication and Reduction:** The modular operation can be substituted with another for loop. A different internal value P(2) (=0 at the beginning) multiplies itself, plus the first incoming values (C or P(1)) if the particular bit of the second incoming value (always P(1)) is 1. In this case, P(1) is checked bitwise from left to right.

Here there is a very important point to highlight, that clearly shows the difference between programming hardware and software:
As the Figure 4.1 suggests, with a software approach, the algorithm from Figure 4.1b

should go inside the algorithm from Figure 4.1a. But for the hardware approach the time needs to be taken into account, so one of them cannot be just disabled while the other one is working. Thus, a different way of resolution must be sought.

In the same way, it is extremely important to discover which sub-steps can be skipped in order to save time in the final performance.

4.2 Block diagram and Micro-architecture

The previous algorithms have been modelled into different blocks, creating the following block diagram:

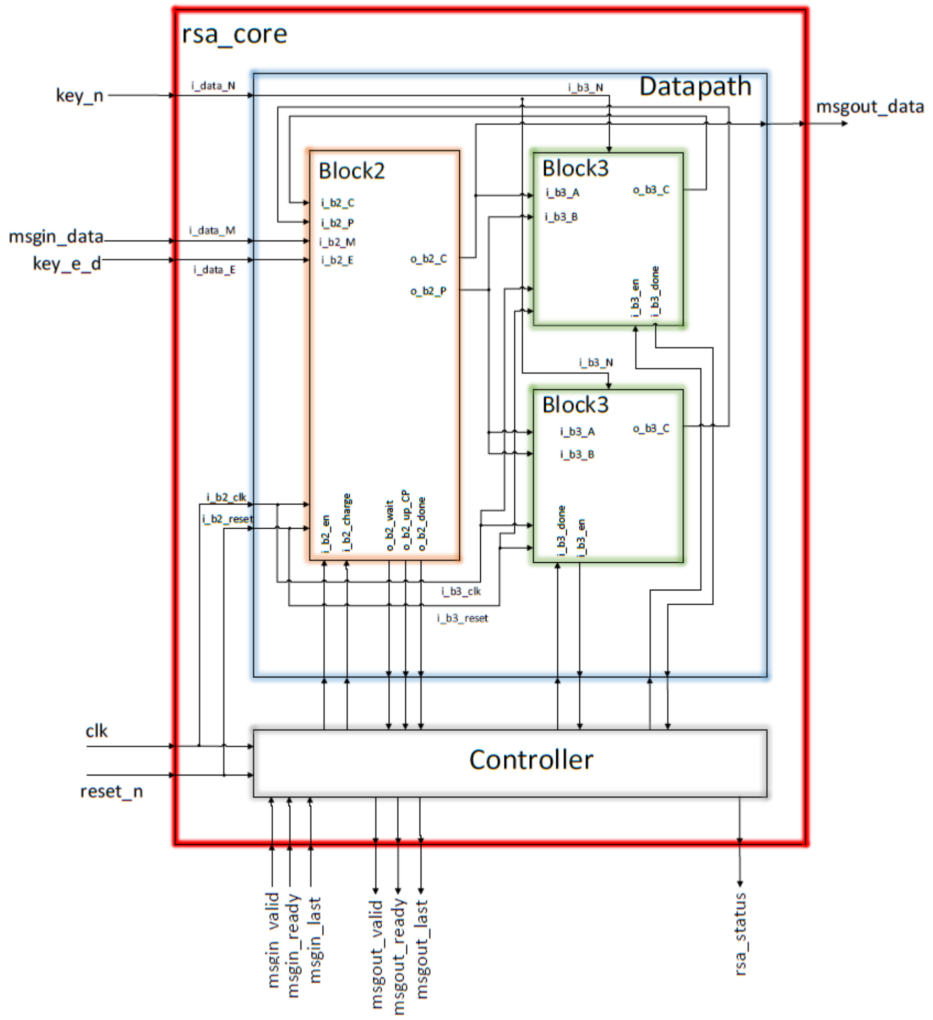


Figure 4.2: Block diagram

This diagram leads to the following micro-architecture:

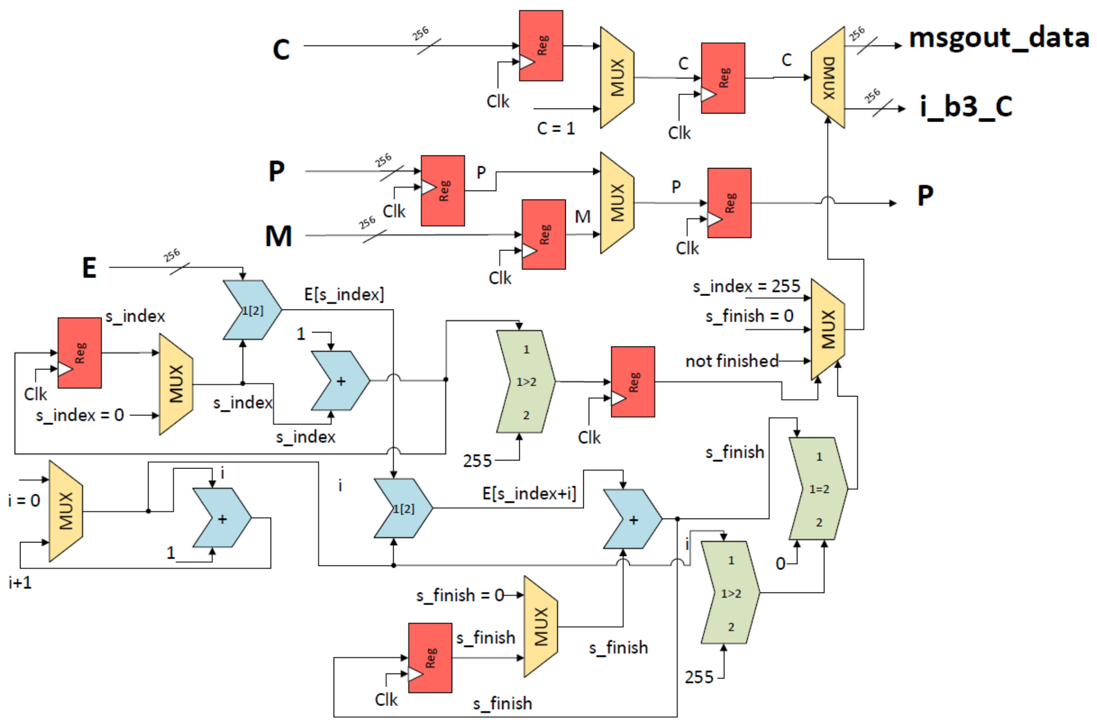


Figure 4.3: Micro-architecture of RL Binary method

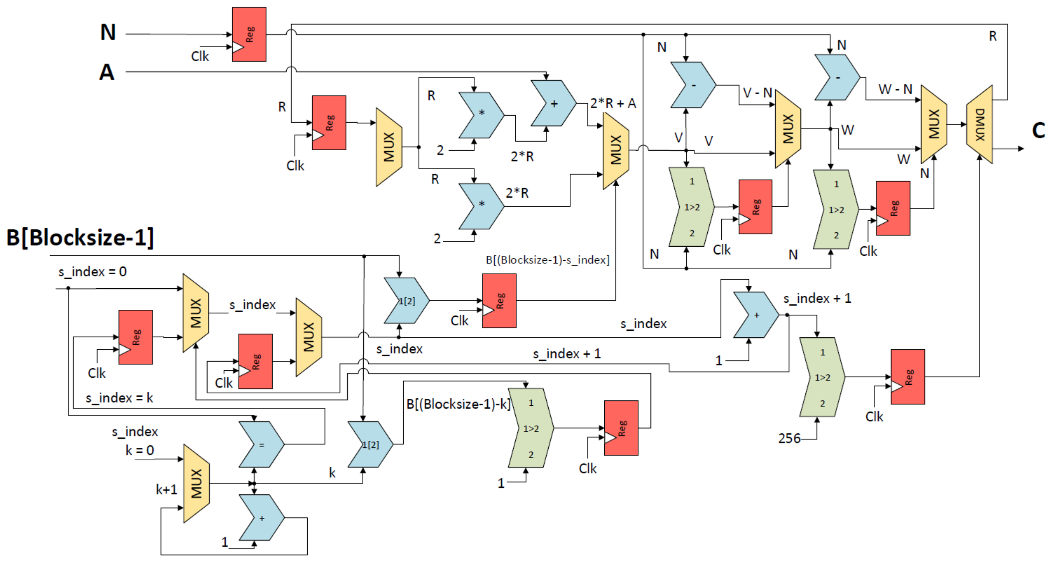


Figure 4.4: Micro-architecture of Interleaving method

These 2 pictures show a quite complex system, and this is because the time component has to be taken into account in HDL designing. In other words, because of this, the two algorithms cannot be integrated one inside another as the original approach suggests so.

4.3 Estimate performance and area

Now there is not enough information to make an estimation of the performance and area of the design, as the lack of experience does not allow us to understand the number of FFs (registers) and LUTs that will be necessary.

Once the design is finished, Vivado can provide these numbers in order to see how efficient and fast the chosen algorithm is. Please refer to subsection nº 7.3.1 to see it.

4.4 Discuss/Analyze/Conclude

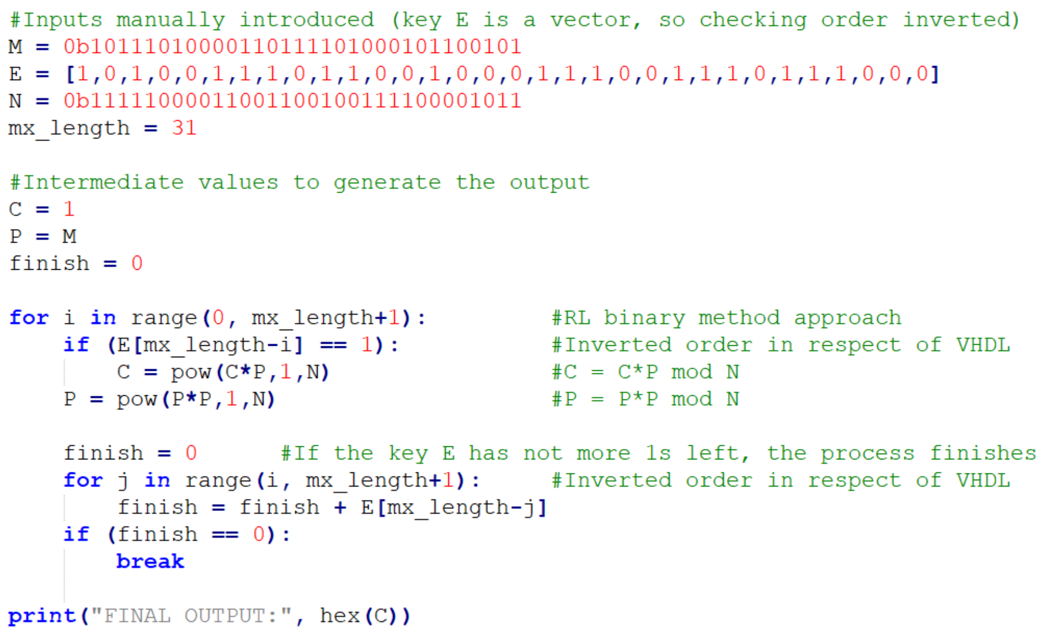
Even though the implementation of the design has not even started so far, for us the most important part of the project is done, which is selecting the proper algorithm to work with.

On the other hand, having a look to the resultant micro-architecture, we expect that the implementation in VHDL is going to be much more complicated as with the typical high-level programming language we are used to.
This can be seen in the previous micro-architecture, where the two selected algorithms cannot be integrated together (as they should go), but they must go separated instead.

5 Implementation of the solution

5.1 Python code

In order to understand better how to proceed with the VHDL implementation, it is highly recommended to do it first with a high level programming language, see figure 5.1. In this way it would be easier to understand what to do, and more important, it can be used later to compare the final result as well as the intermediate ones.



```python
#Inputs manually introduced (key E is a vector, so checking order inverted)
M = 0b10111010000110111101000101100101
E = [1,0,1,0,0,1,1,1,0,1,1,0,0,1,0,0,0,1,1,1,0,0,1,1,1,0,1,1,1,0,0,0]
N = 0b11111000011001100100111100001011
mx_length = 31

#Intermediate values to generate the output
C = 1
P = M
finish = 0

for i in range(0, mx_length+1):          #RL binary method approach
    if (E[mx_length-i] == 1):             #Inverted order in respect of VHDL
        C = pow(C*P,1,N)                  #C = C*P mod N
    P = pow(P*P,1,N)                      #P = P*P mod N

    finish = 0      #If the key E has not more 1s left, the process finishes
    for j in range(i, mx_length+1):       #Inverted order in respect of VHDL
        finish = finish + E[mx_length-j]
    if (finish == 0):
        break

print("FINAL OUTPUT:", hex(C))
```

Figure 5.1: RSA encryption/decryption algorithm in Python

Figure 5.1 shows the implementation of the requested design in Python. As for now the computational time is not important (it is only for testing purposes), the second algorithm "Interleaving Multiplication and Reduction" (Figure 4.1b) can just be skipped. Instead, it is substituted by the high level function "pow", making programming easier. Input messages of 32 bits are used in this code to make the process more readable.

Furthermore, this code also implements an additional improvement to save some computational time by means of an extra for loop (not important now as said before, but for VHDL it will be).
In the section nº 5.4 the whole functionality is explained.

NOTE: As a matter of fact, one can notice that the Python code checks the input key E from left to right when it should be from right to left. This is because the other inputs are just single values (so the LSB is on the right) when E is an array instead (so the LSB is on the left). Thus, the pointer must go in this case in the opposite direction.

5.2 External interface of RSA core

As commented before, for this project only the core of the RSA cryptosystem must be implemented, which means the input message and keys are given, already with the mathematical relations they must have in respect of each other (coprimes).
Thus the RSA core needs to communicate with an external interface to take such data, and the way to do so is described in the following timing diagram:

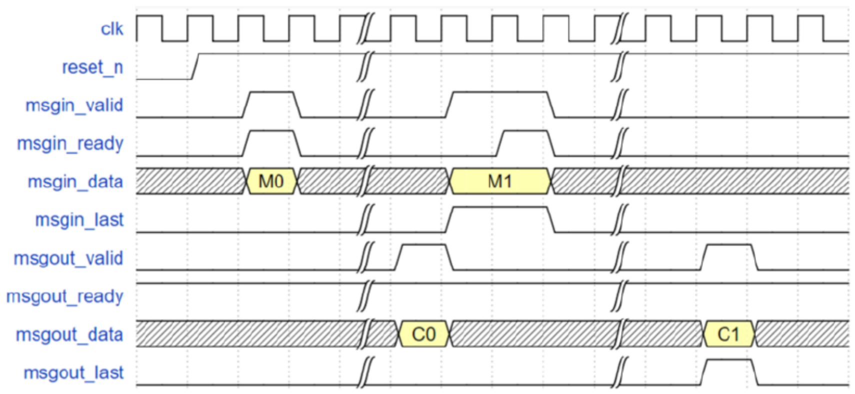


Figure 5.2: Input and output messages interface of RSA core

The sender informs to the core that it is prepared with the message msgin_valid. Once the core reads such a flag, it can take the input data and confirm it has been done with the flag msgin_ready (which must last just one clock cycle).

In the same way, once the core finishes, it informs to the receiver with the flag msgout_valid, and the receiver will confirm the output result has been taken with the flag msgout_ready.

For both cases, the extra message msgin_last (and msgout_last from the core's side) is set when the incoming data is the last chunk of data.

5.3 VHDL code/testbench

The VHDL code (all modules with their respective individual testbench) are attached at the end of the document, in the chapter n^{o} 9, called Appendix.

This functionality has been programmed using a Top-Down approach: From the high-level perspective into a low-level perspective. This designing approach eases the task by defining at first the necessary modules to perform the correct algorithm, and then make them work independently (even helping the designers allowing parallelism work).

As a good practise in VHDL, all modules have one of these possible structures:

```
process (clk, reset_n) begin
  if(reset_n = '0') then
    a_r <= (others => '0');
    b_r <= (others => '0');
  elsif(clk'event and clk='1') then
    if(input_reg_en ='1') then
      a_r <= a_nxt;
      b_r <= b_nxt;
    end if;
  end if;
end process;
```

(a) Template for registers

```
process (data_in, a_r, b_r) begin
  a_nxt <= data_in & a_r(127 downto 32);
  b_nxt <= a_r(31 downto 0) & b_r(127 downto 32);
end process;
```

(b) Template for combinational logic

Figure 5.3: Templates for programming in VHDL

That prevents the creation of latches and ensures perfect synchron. among all modules.

5.4 Solution description

The main module, called RSA_CORE, is composed by 2 blocks: DATAPATH (in charge of doing all necessary calculations) and the CONTROLLER (in charge of managing the DATAPATH). This design is very common in VHDL design as it is a standardized approach to face big systems.
In the same way, the module DATAPATH is divided into 2 modules: BLOCK 2 (algorithm "RL binary method") and BLOCK 3 (algorithm "Interleaving Multiplication and Reduction"). BLOCK 3 is duplicated inside of the DATAPATH module in order to perform the following formulas with parallelism, so DATAPATH is actually divided into 3 modules.

$$C = (C \cdot P) mod N \qquad and \qquad P = (P \cdot P) mod N \tag{5.1}$$

Going now deeper in the functionality itself:

- **BLOCK 2:** With the maximum priority, the reset signal restarts the whole module if activated (negative logic), and if deactivated, the rest of the code executes in every rising edge of the clock signal.
  When the enable signal is 1, values are reinitialized if charge signal is 1 (for repeating the whole process without an external reset). Otherwise the new C and P are reloaded, preparing BLOCK 3 to work with the new data.

After all (re)initializations, the next bit, from right to left, of key E is checked (so the main loop is as long as the key E is). If it is 1, both P and C are updated and requested to calculate the new value, but if it is 0, only P is processed. Values will not be updated in the first iteration with E[i] = 1 because they have not been calculated so far (so incoming value still 0).

Finally, as the final output (C) is only updated when E(i)=1, after every iteration the rest of positions are checked with a for loop. If there are no more 1s, the process finishes there, with the last value of C as the final result.

- **BLOCK 3:** As BLOCK 2, with the maximum priority, the reset signal restarts the whole module if activated (negative logic), and if deactivated, the rest of the code executes in every rising edge of the clock signal.
  When the enable signal is 1, a loop as long as B is done, processing

  $$P(2) = 2 \cdot P(2) + A \cdot B[i]. \tag{5.2}$$

  As P(2)=0 at the beginning, one can notice that P(2) would remain as 0 until the first B[i]=1 (B[i]=0 just removes A too). With this idea on mind, before doing any operation, there is a for loop to discover which position holds the first 1 in B. Then that would be the starting point for the pointer called index.

- **DATAPATH:** This module basically connects BLOCK 2 and two BLOCK 3 (for paralleled computing). Its outputs are (apart from the final output result) many flags from the previous modules, that will be used by CONTROLLER to determine which module should work and which one should wait (the signals enable, inputs in this module).

- **CONTROLLER:** This module manages DATAPATH through a Finite State Machine (FSM), but it is implemented with 2 processes: the second one (not clocked) uses the inputs and intermediate wires to define the FSM itself, and the first one (clocked) uses such wires to apply the value into registers, which are the outputs.

  One more time with the maximum priority, the reset signal restarts the whole module if activated (negative logic), and if deactivated, the rest of the first process executes in every rising edge of the clock signal. The second one executes which any modification in the registers and inputs.

  This module includes the signals from the main module RSA_CORE: "msgin_valid", "msgin_ready", "msgin_last", "msgout_valid", "msgout_ready", "msgout_last". They are used to synchronize the present project with the exterior components that must work with, to implement the whole RSA cryptosystem.

The states are:

1. **IDLE:** Stand-by state. Waiting for initialization (msgin_valid = 1).
2. **CHARGE_VALUES:** Values in BLOCK 2 are loaded (msgin_ready = 1).
3. **WORK_B2:** Only BLOCK 2 (main loop) is allowed to work.
4. **WORK_B32:** Only one BLOCK 3 (for updating P) is allowed to work.
5. **WORK_BOTH_B3:** Only both BLOCK 3 (for updating C and P) are allowed to work.
6. **FINISHED:** The process is finished. Waiting for the next request (msgout_valid = 1 and msgout_ready = 1).

The FSM can be understood better with the following diagram:

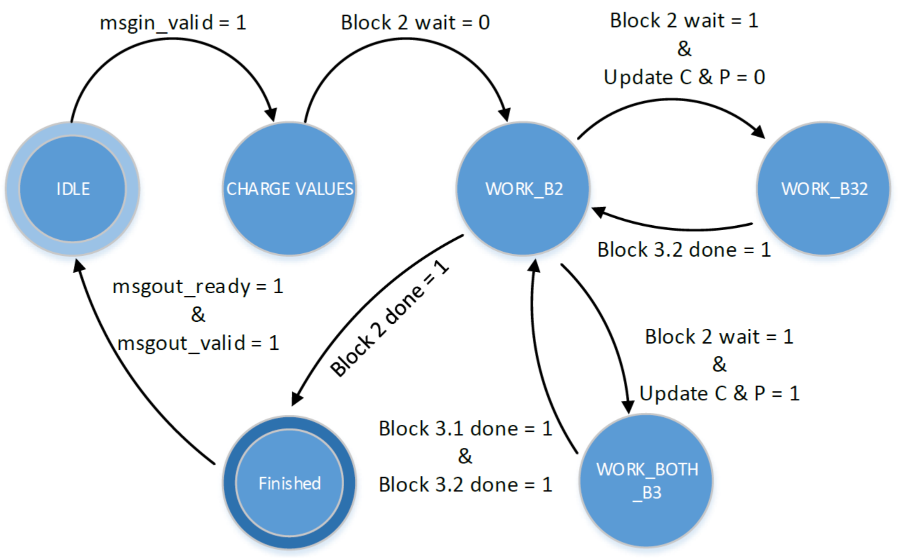

msgin_valid = 1
Block 2 wait = 0
Block 2 wait = 1 & Update C & P = 0
IDLE
CHARGE VALUES
WORK_B2
WORK_B32
Block 3.2 done = 1
msgout_ready = 1 & msgout_valid = 1
Block 2 done = 1
Block 2 wait = 1 & Update C & P = 1
Block 3.1 done = 1 & Block 3.2 done = 1
Finished
WORK_BOTH_B3

Figure 5.4: FSM diagram

In addition, all the design has been prepared to work with a variable message/key size. All VHDL files (except CONTROLLER, which only works with flags), have one constant value ("generic" statement called C_BLOCK_SIZE) that defines how long the chunks of incoming messages are going to be. By default this value is 256 (according to the requirements), but the design will work with any other input size if the value of this constant is changed in all requested files.

5.5 Discuss/Analyze/Conclude

The implementation was more difficult than expected (especially the commun. between modules), and many assertions were required to ease the VHDL simulations, sometimes quite annoying. The Python code was an excellent help to check all intermediate values.

6 Verification plan

6.1 Proposal of the verification plan

Every single module programmed with VHDL (including the modules that just connect the other ones internally) has its own testbench, so that every one of them can be verified in a independent way.
To check all calculated numbers are correct, the small code programmed with a high-level language (Python) is also used to compare all results (not only the final one, but the intermediate ones too).

Although the design has been done using a Top-Down approach (from high-level to low-level), the verification must be done in the opposite direction, and that means from the lowest-level modules (BLOCK 2 and BLOCK 3) up to the highest-level one (DATAPATH, CONTROLLER and RSA CORE). This needs to be done like this because the lower-level modules have been thought to be independent each other, so any change on them affects to the higher-level ones, but not in the other way around.

Anyway, once everything is finished, a complete test of every module will be done one more time, independently that it was also done before with a positive result.

6.2 VHDL simulations

Before programming the Zynq board, some simulations are done of every module in order to verify the good functionality. To make the results more readable, inputs messages (M, E and N) of 32 bits have been used here:

- Figure 6.1 shows the simulation of BLOCK 2. It can be seen that the input values are charged when both reset (negative logic) or charge signal are set, and then the new input value P is charged in the internal variable (but not in the first iteration). On the other hand, the new input value C is charged internally only when o_b2_up_CP(E[i])=1. The index at the bottom shows the iteration number.

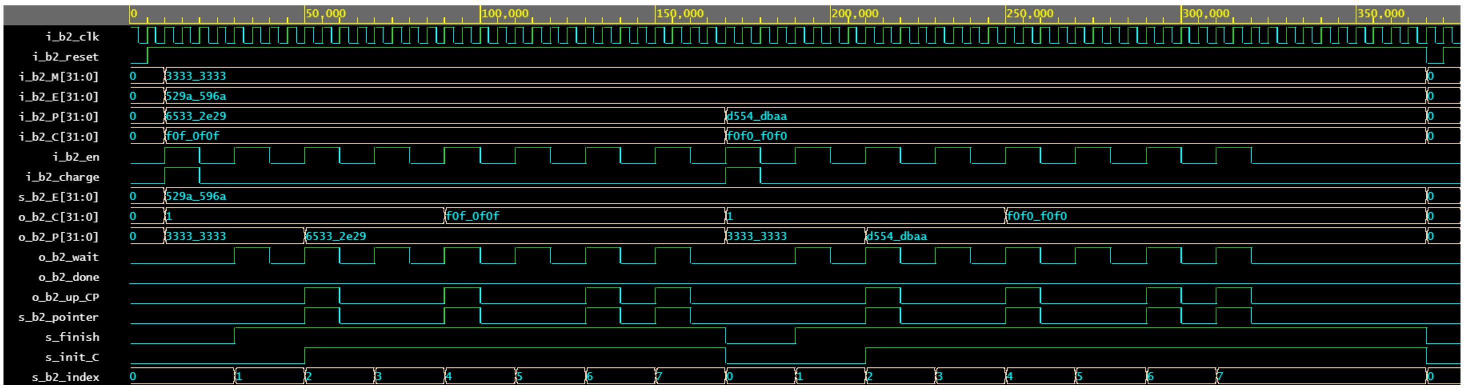


Figure 6.1: Simulation of Block 2

- Figure 6.2 shows the simulation of BLOCK 3. This module performs the module operation given some input values. It can be seen that a new output is calculated in every iteration, up to 32 times (as input values are 32 bits long), when signal done is set. Here it is very important to cross-check the final result with the previously code implemented with Python (equivalent to 1 iteration in the complete design).

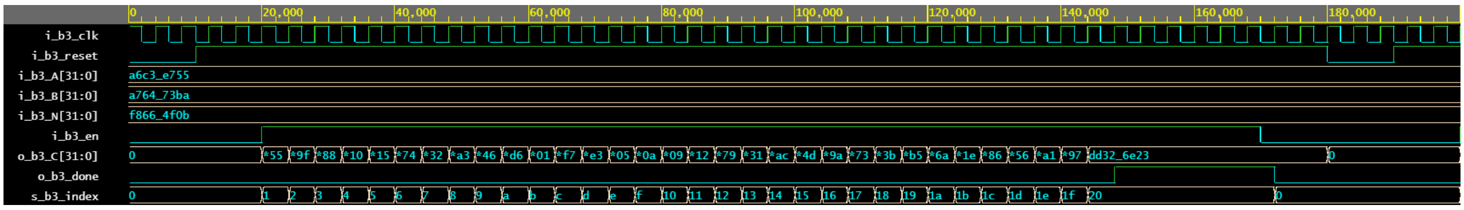


Figure 6.2: Simulation of Block 3

- Figure 6.3 shows the simulation of DATAPATH. This module integrates BLOCK 2 and (twice) BLOCK 3. In this particular simulation, up to 5 iterations of the complete design can be checked, which can be cross-checked with the Python implementation to know if the design is correct so far.

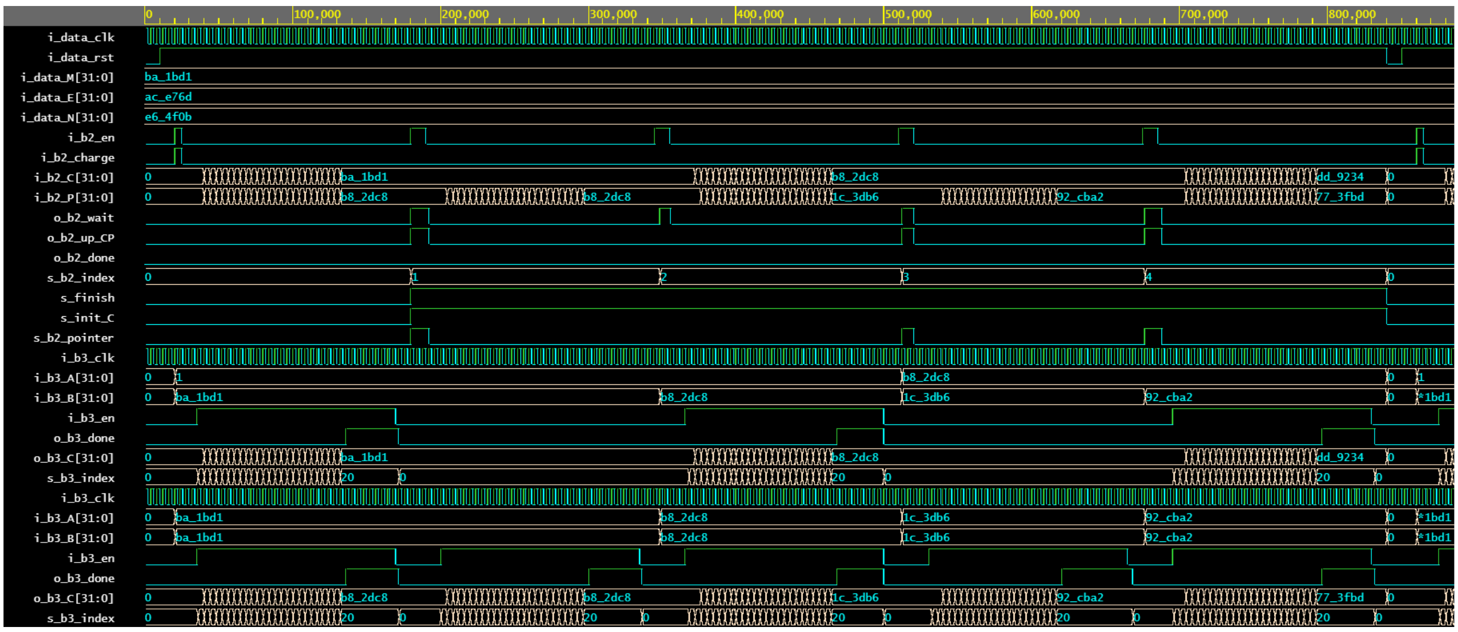


Figure 6.3: Simulation of Datapath

- Figure 6.4 shows the simulation of CONTROLLER. This module manages the FSM that controls the DATAPATH. Given the external inputs msgin_valid, msgin_last and msgout_ready, the FSM moves through all states (displayed at the bottom as o_fsm_state) and it is continuously modifying the enable outputs that go to DATAPATH, to control the either BLOCK 2 or BLOCK 3.

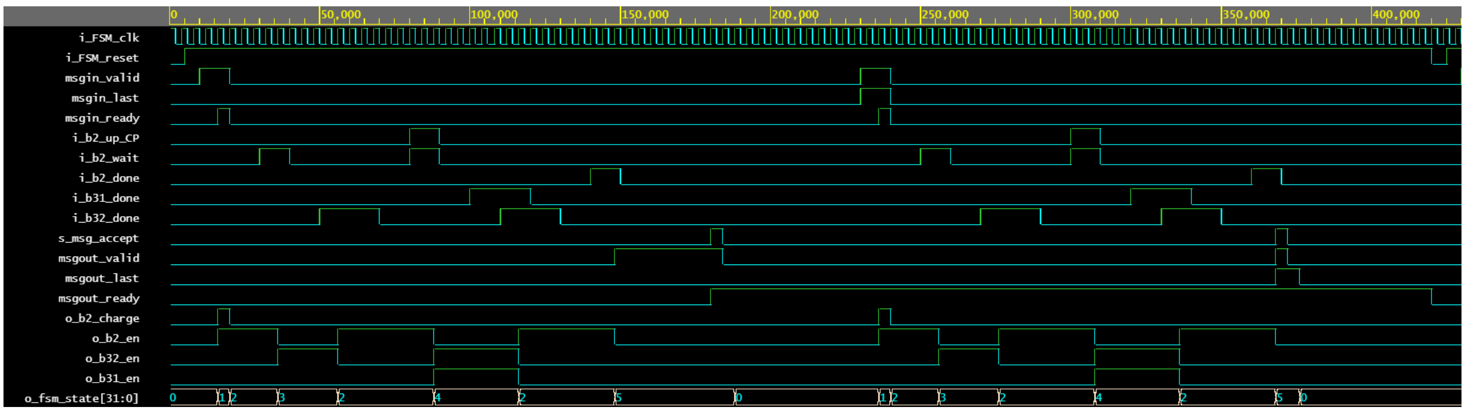


Figure 6.4: Simulation of Controller

- Figure 6.5 shows the simulation of RSA CORE. Integrating every module explained above, every single step can be checked here. Two steps are performed: one not being the last message and another one it is (simulating a 512 bit input).

Here the intermediate values and the final value can be cross-checked with the Python implementation. If everything is ok (as it is now), one assumes the final designed is validated.

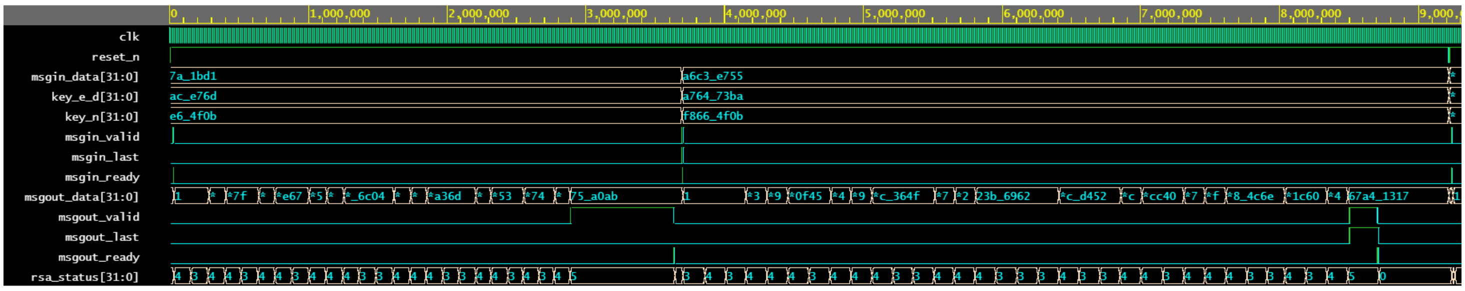


Figure 6.5: Simulation of RSA core (main module)

6.3 Metrics to decide when verifying is over

Once the final design in VHDL matches with the results in Python, the verification will be finished if they also match with an online RSA encryption/decryption tool.
But in case both the public key (E) and (D) are known, the best practise would be to encrypt the message, and then decrypt it. If the final output is the same as the initial input, then the process is perfect.

Last but not least, now probably as a redundant test, the very final test will be handled with professor's testbench, which actually should be very fast to do if all previous steps were done accordingly.

6.4 Using assertions

During the designing phase, the modules BLOCK 2, BLOCK 3 and CONTROLLER (the rest just interconnect other modules) include many signals to help during the testing phase.

From counters, boolean flags and even intermediate values working as variables (which means they had to be saved in signals too in order to be possible to check) are added in the respective designs, not only as important data to share between modules, but also to provide a clear vision about in which step the algorithm currently is.

Furthermore, it was decided to update all intermediate values as well as the final result after every iteration and not only at the end of the local algorithm. Only with this approach can every single new value be cross-checked with the Python code properly,

6.5 Discuss/Analyze/Conclude

Surprisingly, once we made all own tests work satisfactory (even comparing results with Python code and the online RSA encryption/decryption tool), we faced problems to pass the very last test with professor's testbench.

The external modules working with RSA_CORE have a strict functionality that we did not design properly, so some synchronization issues with the inputs and outputs msgin_valid, msgin_last, msgin_ready, msgout_valid, msgout_last and msgout_ready had to be fixed.
This problem made us understand better the concept of the sensitivity list and the importance to do some operations outside the processes as well.

7 Synthesis and test on FPGA

7.1 PYNQ-Z1 Board

The development board used for this project is a PYNQ-Z1 ZYNQ-7000 FPGA board. The system-on-chip is running Linux as operating system. It is possible to access the board via a browser based computing environment, called Jupyter Notebook. There, live code can be written and tested on the board, using the programming language Python. The development board contains a SD-Card, where all needed files for the operating system, as well as the hardware design files are stored.

7.2 Synthesis of the RSA_CORE

After testing the written VHDL code by using testbenches for every single created module, the whole RSA_CORE was put together in a Vivado project folder. This project folder served the purpose of testing, if all the modules are working flawlessly together.
The teaching staff provided a testbench, which could be used to analyze the behaviour of the RSA_CORE. It uses the interface of the RSA_CORE with its surrounding system, which was already mentioned in 5.4. Besides the testbench, some test messages have been provided, which are going to be send to the RSA_CORE. To verify if the provided text messages are successfully encrypted, a check of the encrypted messages is done by comparing the encrypted message with an given expected value. This expected value was computed beforehand and therefore already provided.

However, this testbench does not guarantee that the developed VHDL code is without errors. After successfully running the testbench, it was necessary to analyze the code for errors by using the synthesis tool of Vivado.
The synthesis tool turnes the VHDL code into a design on logic gate level, thus, it creates the hardware model. During this process, warnings are given if the code infers latches, signals have no drivers and many other possibilities for errors. These errors might not be a problem during simulation, because the simulation is mainly based on the input and output signals, as well as signal delays.

Especially inferred latches should be avoided, given that latches can create unexpected behaviour, because they are driven asynchronously. Simulations might run successfully, even though inferred latches exist in the code. But hardware, such as FPGAs, use clocked designs, which means every flip flop is clocked by the same clock and therefore driven synchronously.

After analyzing the message log of the synthesis and removal of inferred latches and possible other warnings, the RSA_CORE can be implemented into the RSA_ACCELERATOR.
The RSA_ACCELERATOR, figure 7.1, consists of four different modules. The module RSA_REGIO module receives the keys for encryption and decryption from a master of the surrounding system. The register addresses, where the keys are stored, are predefined by the teaching stuff and have not been changed. The RSA_MSGIN receives the input data of the messages, which shall be encrypted. Because the input channel is limited to 32 bits, this module acts as a serial to parallel converter. When eight 32 bit messages have been received, the module will create one 256 bit message, which will be send to the RSA_CORE for encryption. RSA_MSGOUT takes the encrypted messages and creates 32 bit chunks out of the 256 bit data.

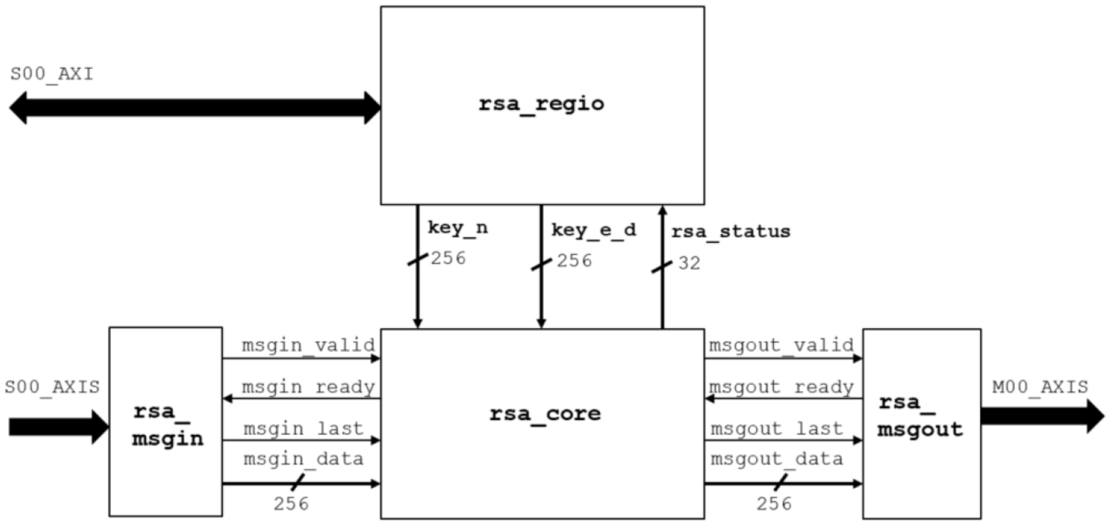


Figure 7.1: The RSA_ACCELERATOR and its components.

The modules surrounding the RSA_CORE are using the AXI[1] interface, which is a microcontroller bus for interconnecting functional-blocks in systems-on-chip (SOC).
To implement the RSA_CORE into the RSA_ACCELERATOR the VHDL files of the core have to be included into its project folder. After packaging the modified RSA_ACCELERATOR, to create a new IP[2], the RSA_CORE is implemented into the final project.
The final project inlcudes all needed files for implementing the code onto the FPGA. In this stage of the project, the synthesis tool of Vivado had to be run for creating the final implementation of the RSA encryption project.

[1]For further information see [3].
[2]IP stands for Intellectual Property and means, that the designer uses existing software components, written by someone else.

The synthesis gives a first hint, if the project with all its code can be build and turned into a design on logic level. However, the final step is the implementation. The implementation creates a hardware design based on the , which can be used on the FPGA. It also states the final amount of flip flops, as well as LUTs, which are needed for the design. Besides the needed area, it is also possible to check if the design meets the timing constraints.

7.3 Implementing the RSA Encryption Project on the FPGA

7.3.1 Area and Performance

After running the synthesis and the implementation of the RSA encryption project, it was necessary to adjust the timing constraints. The initial timing constraints were set to a clock period of $T = 10\ ns$ which equals a frequency of $f = 100\ MHz$. The timing report of the implementation shows the "Worst Negative Slack" (WNS). This time gives therefore the additional needed time for the critical path[3] to reach its endpoint.
This timing constraint could not be met with the RSA_CORE. By adding the WNS to the set period the new clock period can be calculated. The clock frequency of the FPGA can be changed via the IP Integrator in Vivado.

As already mentioned before, Vivado allows to get information about the used area of the FPGA and the performance. The ZYNQ-7000 contains the following resources listed in table 7.1.

Resource	Utilization	Available	Utilization %
LUT	10213	53200	19.2
LUTRAM	603	17400	3.47
FF	9087	106400	8.54
BRAM	2	140	1.43

Table 7.1: Used area on the FPGA

As can be seen in the table, the RSA encryption uses only 19.2% of the available LUTs and 8.54% of the available FFs. This means, that it would be theoretically possible to implement the whole project five times on the FPGA to achieve a higher throughput of messages.

The design can run on a frequency of $f = 34.48\ MHz$, thus $T = 28.99\ ns$. This leaves a positive slack of 0.315 ns. The critical path of the design has a positive slack of 0.315 ns. It is located in the DATAPATH module and goes from Block 2 from signal s_b2_P_reg[206] to Block 3 to variable v_b3_C_reg[256].

[3]The critical path is defined as the path with the greatest delay between two registers.

The total on-chip power is around 1.43 W. The figure 7.2 shows a bar diagram, which is calculated by Vivado. It shows the dynamic and the static power of the chip.

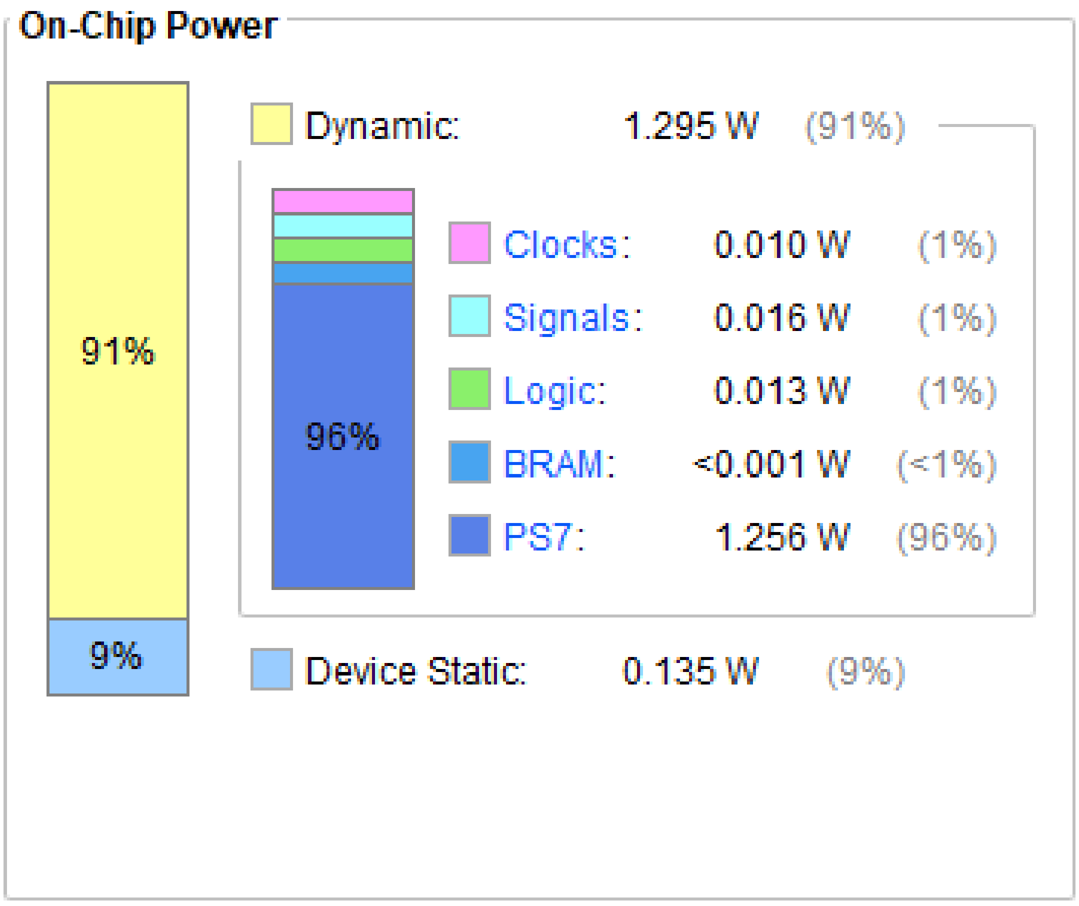


Figure 7.2: Power consumption of the chip.

7.4 Test of the Design

7.4.1 Implementation on the FPGA

After analyzing all information and adjusting the timing constraints, the need files for the hardware are generated. A BIT file and a TCL file are needed. The BIT file contains the hardware design and the TCL file contains information about the BIT file. To implement the created implementation of the VHDL code on the FPGA, these two files have to be stored on the SD-Card of the development board and are accessed and used later via the live code on the Jupyter Network.

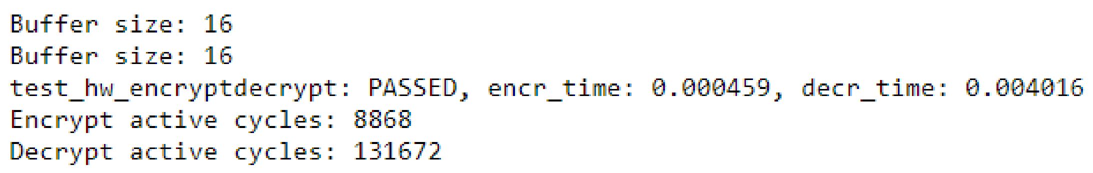
A test environment is given for the Jupyter Network. This live code is written in Python and gives the opportunity to test the created RSA encryption design in real time on the FPGA. In the following passage a brief overview of what the live code does, is given.

The code sets up the RSA encryption and loads the messages which shall be encrypted and decrypted. After loading the messages, the keys N, E and D are loaded. The DMA of the FPGA is set up for reading the messages from the memory and sending them to RSA_CORE. Now, the loaded keys are sent to the registers of RSA_REGIO module.

The next step of the code is the encryption and decryption of the messages by using the hardware. The last part gives information about the performance of the hardware design in comparison to a high-level software language.

7.4.2 Results of Encryption and Decryption

The test run of the hardware was successful. Figure 7.3 shows the encryption and decryption time as well as the needed clock cycles of the hardware. The time is given in seconds, which means that the encryption needed $t_{encrypt} = 459\ \mu s$. The decryption needed $t_{decrypt} = 4.016\ ms$.



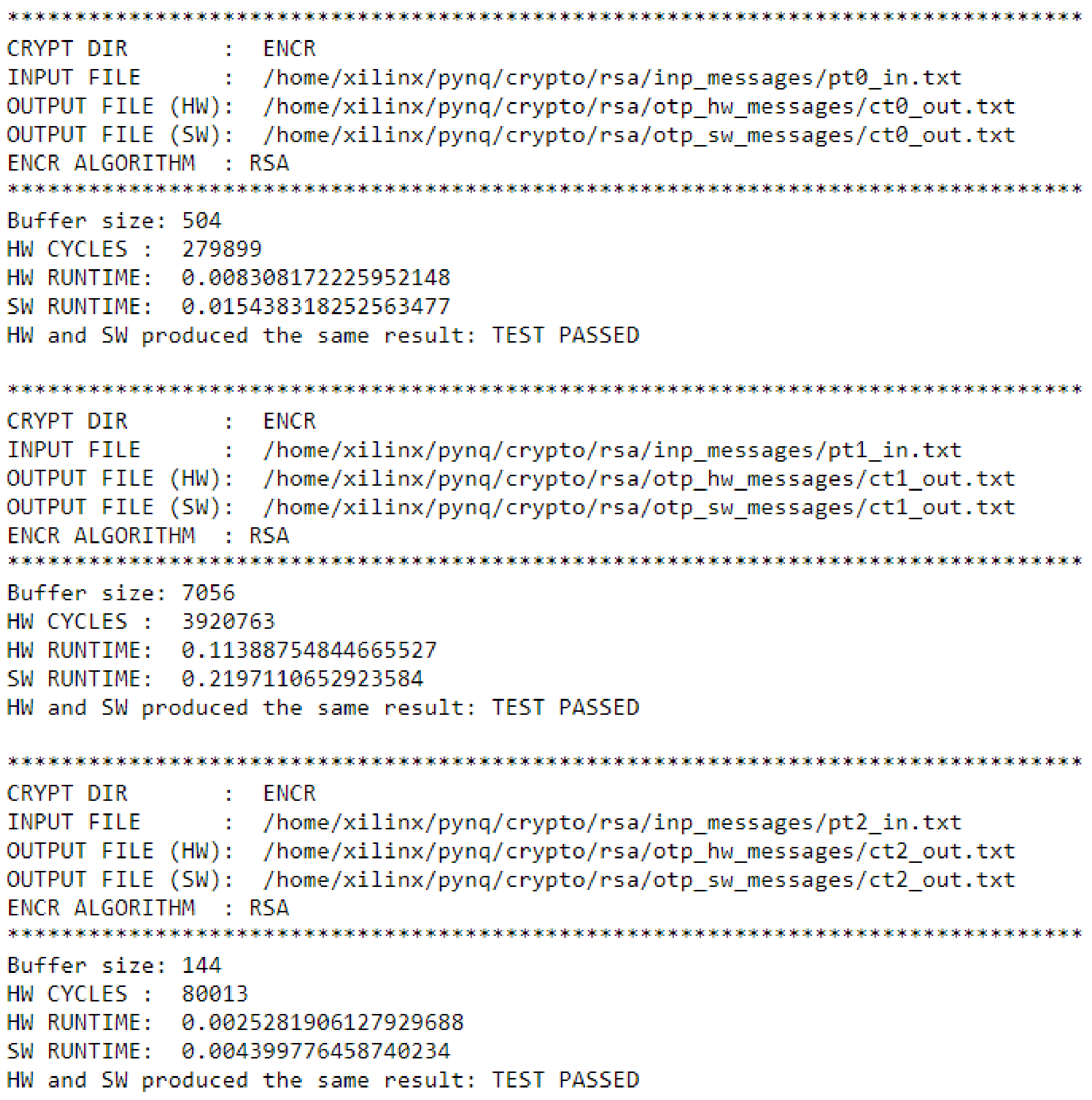
```
Buffer size: 16
Buffer size: 16
test_hw_encryptdecrypt: PASSED, encr_time: 0.000459, decr_time: 0.004016
Encrypt active cycles: 8868
Decrypt active cycles: 131672
```

Figure 7.3: Encryption and decryption time and needed clock cycles of the hardware.

Very noticeable is the difference in time and clock cycles needed. The RSA process for encryption and decryption is the same, but uses different keys, as mentioned in section 3.3. This difference comes from the fact that the keys are checked every iteration for bits which are 1. If all remaining bits of the key are 0, the process ends. This is explained in section 5.4.
By comparing the used encryption and decryption keys it is obvious why the difference is this noticeable:

- key_e = 0x0000000000000000000000000000000000000000000000000000000000010001
- key_d = 0x0cea1651ef44be1f1f1476b7539bed10d73e3aac782bd9999a1e5a790932bfe9

The second test run with long messages was also successful. The following figure, 7.4, shows the comparison of the hardware encryption time and the software encryption time.



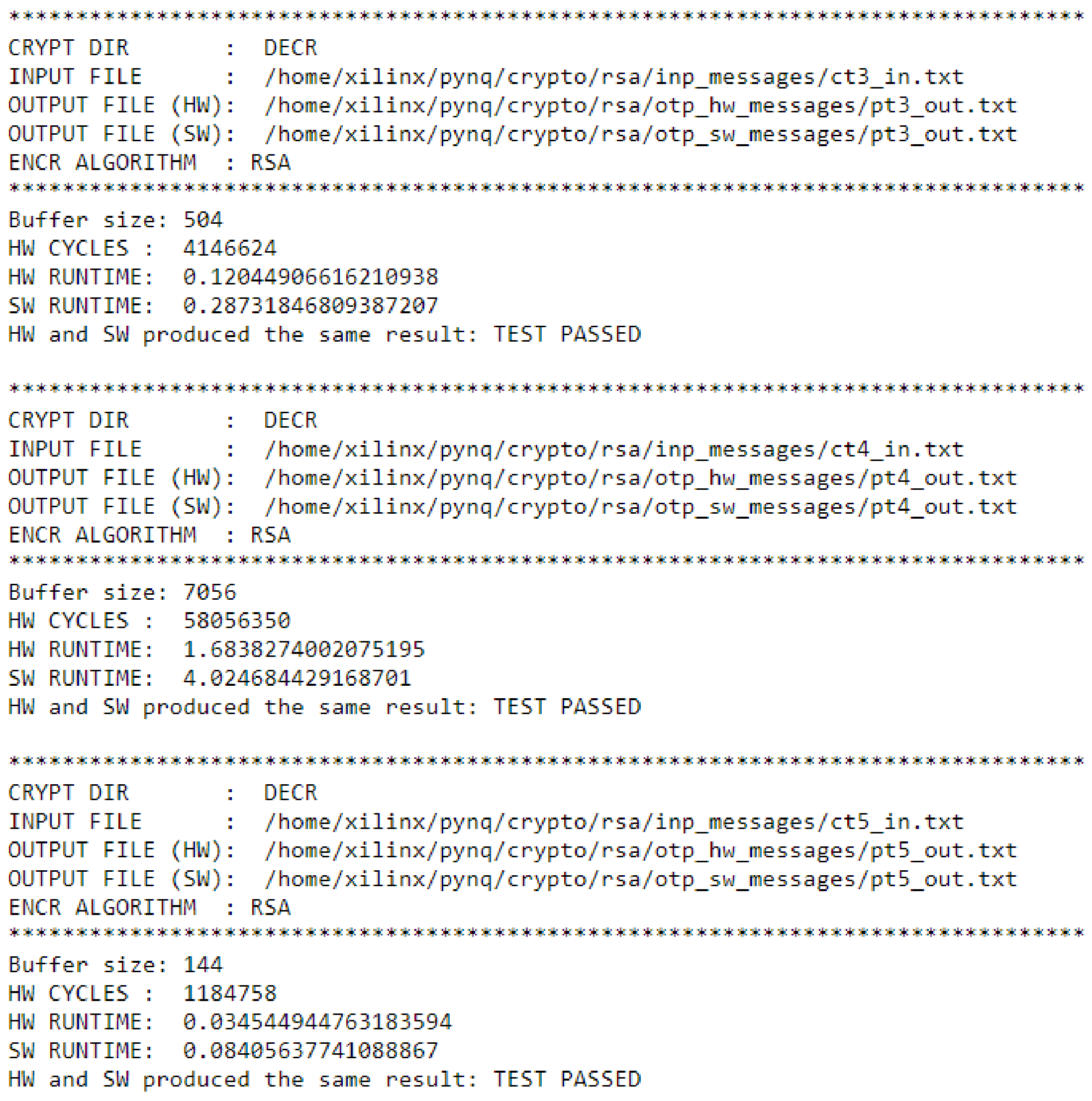
```
********************************************************************************
CRYPT DIR       :  ENCR
INPUT FILE      :  /home/xilinx/pynq/crypto/rsa/inp_messages/pt0_in.txt
OUTPUT FILE (HW):  /home/xilinx/pynq/crypto/rsa/otp_hw_messages/ct0_out.txt
OUTPUT FILE (SW):  /home/xilinx/pynq/crypto/rsa/otp_sw_messages/ct0_out.txt
ENCR ALGORITHM  : RSA
********************************************************************************
Buffer size: 504
HW CYCLES :  279899
HW RUNTIME:  0.008308172225952148
SW RUNTIME:  0.015438318252563477
HW and SW produced the same result: TEST PASSED

********************************************************************************
CRYPT DIR       :  ENCR
INPUT FILE      :  /home/xilinx/pynq/crypto/rsa/inp_messages/pt1_in.txt
OUTPUT FILE (HW):  /home/xilinx/pynq/crypto/rsa/otp_hw_messages/ct1_out.txt
OUTPUT FILE (SW):  /home/xilinx/pynq/crypto/rsa/otp_sw_messages/ct1_out.txt
ENCR ALGORITHM  : RSA
********************************************************************************
Buffer size: 7056
HW CYCLES :  3920763
HW RUNTIME:  0.11388754844665527
SW RUNTIME:  0.2197110652923584
HW and SW produced the same result: TEST PASSED

********************************************************************************
CRYPT DIR       :  ENCR
INPUT FILE      :  /home/xilinx/pynq/crypto/rsa/inp_messages/pt2_in.txt
OUTPUT FILE (HW):  /home/xilinx/pynq/crypto/rsa/otp_hw_messages/ct2_out.txt
OUTPUT FILE (SW):  /home/xilinx/pynq/crypto/rsa/otp_sw_messages/ct2_out.txt
ENCR ALGORITHM  : RSA
********************************************************************************
Buffer size: 144
HW CYCLES :  80013
HW RUNTIME:  0.0025281906127929688
SW RUNTIME:  0.004399776458740234
HW and SW produced the same result: TEST PASSED
```

Figure 7.4: Encryption of long messages. Hardware and software comparison.

Figure 7.5 shows the decryption of messages and the achieved results.



```
********************************************************************************
CRYPT DIR       :  DECR
INPUT FILE      :  /home/xilinx/pynq/crypto/rsa/inp_messages/ct3_in.txt
OUTPUT FILE (HW):  /home/xilinx/pynq/crypto/rsa/otp_hw_messages/pt3_out.txt
OUTPUT FILE (SW):  /home/xilinx/pynq/crypto/rsa/otp_sw_messages/pt3_out.txt
ENCR ALGORITHM   : RSA
********************************************************************************
Buffer size: 504
HW CYCLES :  4146624
HW RUNTIME:  0.12044906616210938
SW RUNTIME:  0.28731846809387207
HW and SW produced the same result: TEST PASSED

********************************************************************************
CRYPT DIR       :  DECR
INPUT FILE      :  /home/xilinx/pynq/crypto/rsa/inp_messages/ct4_in.txt
OUTPUT FILE (HW):  /home/xilinx/pynq/crypto/rsa/otp_hw_messages/pt4_out.txt
OUTPUT FILE (SW):  /home/xilinx/pynq/crypto/rsa/otp_sw_messages/pt4_out.txt
ENCR ALGORITHM   : RSA
********************************************************************************
Buffer size: 7056
HW CYCLES :  58056350
HW RUNTIME:  1.6838274002075195
SW RUNTIME:  4.024684429168701
HW and SW produced the same result: TEST PASSED

********************************************************************************
CRYPT DIR       :  DECR
INPUT FILE      :  /home/xilinx/pynq/crypto/rsa/inp_messages/ct5_in.txt
OUTPUT FILE (HW):  /home/xilinx/pynq/crypto/rsa/otp_hw_messages/pt5_out.txt
OUTPUT FILE (SW):  /home/xilinx/pynq/crypto/rsa/otp_sw_messages/pt5_out.txt
ENCR ALGORITHM   : RSA
********************************************************************************
Buffer size: 144
HW CYCLES :  1184758
HW RUNTIME:  0.034544944763183594
SW RUNTIME:  0.08405637741088867
HW and SW produced the same result: TEST PASSED
```

Figure 7.5: Decryption of long messages. Hardware and software comparison.

Every encryption and decryption process was successful. The implemented design is therefore fully functional.
The times are given in seconds. Noticeable is, that the hardware is always faster than the software.

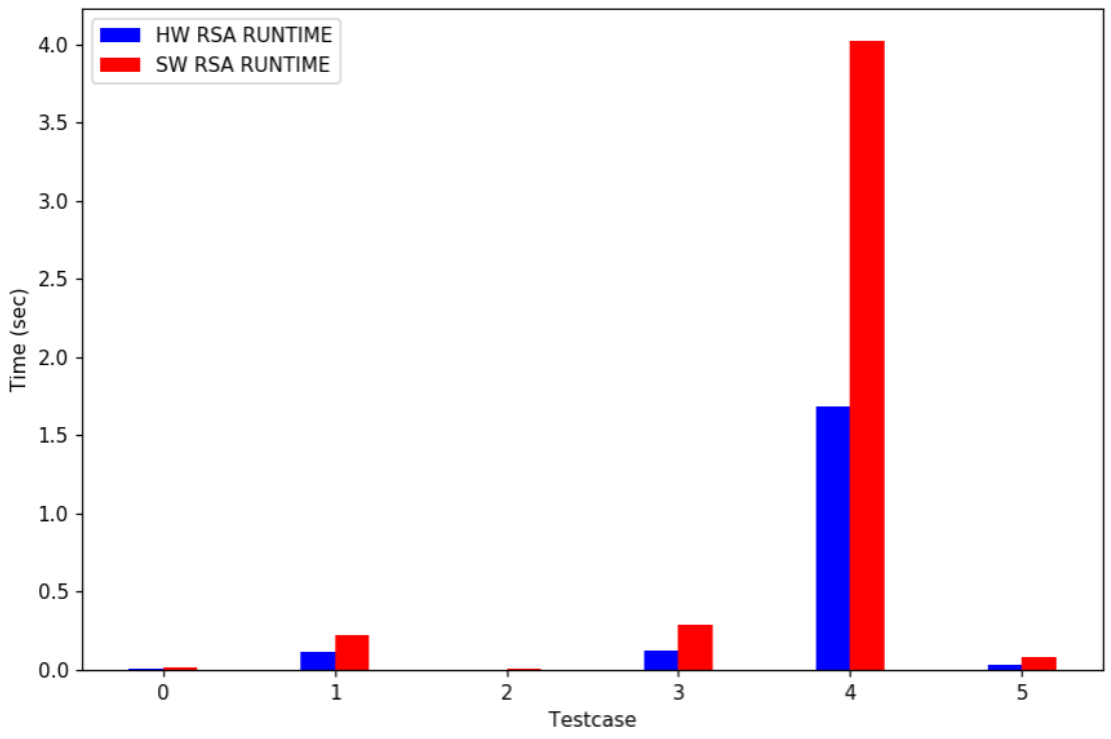


Figure 7.6: Bargraph of the run time of hardware and software.

Figure 7.6 shows a bar graph of the run time of the hardware in comparison to the software.

7.5 Discuss/Analyze/Conclude

Vivado is a very extensive program with a lot of functionalities. Therefore, it was only possible to get a quite basic understanding of this hardware design tool. Nevertheless, given the time and the extent of the project, it was possible to learn a lot about its usage during the design process, while working with the given term_project folder.
The PYNQ-Z1 is an easy to use development board with the possibility to run live code. This makes validation of created designs very convenient.

The implementation of the VHDL code into the RSA_ACCELERATOR was very good manageable if the interface was correctly implemented. However, it was necessary in the beginning to include the designed VHDL files of the RSA_CORE to the IP packaging process of the project. Otherwise, these files were detected as black boxes and could not be accessed by Vivado.

Another problem we encountered while implementing the hardware code onto the FPGA was, that two latches have been inferred. The simulation of the testbench run without problem, after adjusting the ready-valid-handshake between the sender and the receiver, even though the code was inferring the latches.
When trying to run the hardware test, the FPGA got hang-up at some point. The problem could be solved by removing the latches out of the VHDL code. As soon as this was done, the hardware test could be run without any further problems.

The hardware design itself could be successfully implemented. The currently used area of the FPGA leaves much space to implement several cores on the FPGA. This could be done as future work.
Additionally, it could be possible to improve the timing of the circuit. Some attention was put on reducing the needed clock cycles, but it might be possible to find ways to reduce the critical path itself.

8 Conclusions and future work

VHDL is a powerful Hardware Description Language (HDL) to design massive digital systems, much faster and much more efficient than with old techniques, and the improvement is even bigger if one compares it with the analog systems, quite inefficient during design and performance.
Nevertheless, we have discovered that the new approach we had to use to solve the requested problem is quite different from the typical high-level software development tools we are used to. Taking time into account as an additional variable into the equation, makes VHDL more complicated to control properly than Python, C, Java... but it is actually quite normal, as those programming languages are thought to create virtual elements, and the VHDL is thought to create hardware (so physical elements) instead.

The RSA cryptographic algorithm has not been that difficult to implement (but quite time-consuming because of the lack of experience so far in this domain), as at the end there were only two for loops (transformed into if statements with the VHDL approach) to be implemented. Anyway the final result provides a quite powerful cryptosystem that introduce us in the basics of cryptography, a very interesting domain.
We did not have time enough to implement and compare all possible alternative algorithms, to see which fits better, but indeed any of them was enough to understand the capabilities of this new approach for designing.

Furthermore, another very important point is that we did not only focus in the simulation of the requested functionality, but we had also the opportunity to apply it into a real FPGA and to check the difference in performance between simulation and reality. This is very important as well to understand the basics, as we discovered at some point that our first solution was simulated perfectly, but it was not working into the real FPGA because of the latches. There is sometimes big differences between simulations and real tests, and of course the maximum clock frequency is not the same either, always depending on how efficient the algorithm is.

With all this, we can proudly say then that all objectives in this project have been fulfilled, as the final result is completely satisfactory, and it let us understand the hardware designing approach as well as the basic functions of the VHDL, even though we recognize there is much more material to deep in. One semester is not enough to know how powerful the VHDL really is.

Unfortunately, we feel that the project is incomplete because only the core of the RSA algorithm was implemented, leaving the generation of the keys to the professor, a part we think is quite challenging as well, as they must be quite huge and still keep some mathematical relations between them.
Doing the complete implementation would be an excellent point as future work. Probably it would give us a more exact idea about what the industry expects to us at implementing still used algorithms in the industrial market.

Overall, we enjoyed this project a lot and strongly recommend this project for further promotions in order to understand the capabilities of the EDA tools, the real tool, which is current used in the industry to create the new hardware designs we use in our daily lives.

9 Appendix

9.1 Block 2 code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
USE ieee.numeric_std.ALL;   -- For function "To_b2_integer"

entity Block_2 is
    generic (
        C_BLOCK_SIZE: integer := 256  -- Number of bits of the incoming message
    );
    port (
        i_b2_clk:   in std_logic;
        i_b2_reset: in std_logic;
        i_b2_en:    in std_logic;
        i_b2_charge:in std_logic;
        i_b2_M:     in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
        i_b2_E:     in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
        i_b2_P:     in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
        i_b2_C:     in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
        o_b2_P:     out std_logic_vector((C_BLOCK_SIZE-1) downto 0);
        o_b2_C:     out std_logic_vector((C_BLOCK_SIZE-1) downto 0);
        o_b2_wait:  out std_logic;
        o_b2_up_CP: out std_logic;
        o_b2_done:  out std_logic
    );
end Block_2;

architecture Behavioral of Block_2 is
    signal s_init_C:    std_logic;
    signal s_b2_Co:     std_logic_vector ((C_BLOCK_SIZE-1) downto 0);
    signal s_b2_P:      std_logic_vector ((C_BLOCK_SIZE-1) downto 0);
    signal s_b2_wait:   std_logic;
    signal s_b2_up_CP:  std_logic;
    signal s_b2_E:      std_logic_vector((C_BLOCK_SIZE-1) downto 0);
    signal s_b2_done:   std_logic;
    signal s_b2_index:  integer range 0 to C_BLOCK_SIZE;
    signal s_b2_pointer:std_logic;
    signal s_finish:    std_logic;
begin
    process (i_b2_clk, i_b2_reset) is
    variable v_b2_pointer: std_logic;
    variable v_finish: std_logic;
    begin
        if (i_b2_reset = '0') then
            s_init_C <= '0';
            s_b2_Co <= (others => '0');
            s_b2_P <= (others => '0');
            s_b2_E <= (others => '0');
            v_b2_pointer := '0';
            s_b2_pointer <= '0';
            s_b2_wait <= '0';
            s_b2_up_CP <= '0';
            s_b2_done <= '0';
            s_b2_index <= 0;
            s_finish <= '0';
            v_finish := '0';
        elsif (rising_edge(i_b2_clk)) then
            if (i_b2_en = '1') then
                if (i_b2_charge = '1') then
                    s_init_C <= '0';
                    s_b2_Co <= (others => '0');
                    s_b2_Co(0) <= '1';  -- Initial value is 1
                    s_b2_P <= i_b2_M;
                    s_b2_E <= i_b2_E;
                    v_b2_pointer := '0';
                    s_b2_pointer <= '0';
                    s_b2_wait <= '0';
                    s_b2_up_CP <= '0';
                    s_b2_done <= '0';
                    s_b2_index <= 0;
                    s_finish <= '0';
                    v_finish := '0';
                elsif (s_b2_wait = '0') then
                    if (s_b2_index <= (C_BLOCK_SIZE-1)) then
                        v_b2_pointer := s_b2_E(s_b2_index); -- Checks vector E from Right to Left!
                        s_b2_pointer <= v_b2_pointer;
                        if (s_b2_index /= 0) then   -- The new values from Block 3 must not be loaded in iteration 0
                            if (v_b2_pointer = '1') then -- C only updated when E(i)= 1
                                if (s_init_C = '1') then
                                    s_b2_Co <= i_b2_C;
                                end if;
                                s_b2_up_CP <= '1';
                                s_init_C <= '1';    -- Only after a new C is processed in Block 3 with Co = 1, Co will be updated
                            end if;
                            s_b2_P <= i_b2_P;
                        else
                            if (v_b2_pointer = '1') then
                                s_b2_up_CP <= '1';  -- C's notification must be checked at i = 0, just to update it in the next attempt (if necessary)
                                s_init_C <= '1';
                            end if;
                        end if;
                        s_b2_wait <= '1';
                        s_b2_index <= s_b2_index + 1;

                        v_finish := '0';        -- Every time the new values are loaded, the rest of E is checked
                        for i in 0 to (C_BLOCK_SIZE-1) loop
                            if (i >= (s_b2_index+1)) then
                                v_finish := v_finish or s_b2_E(i);
                            end if;
                        end loop;
                        s_finish <= v_finish;
                        if (v_finish = '0') then        -- If there are no more 1s left, no sense to continue
                            s_b2_index <= C_BLOCK_SIZE; -- As output will not change
                        end if;
                    else
                        s_init_C <= '0';
                        s_b2_wait <= '0';
                        s_b2_up_CP <= '0';
                        s_b2_Co <= i_b2_C;  -- The output is updated here just in the very last iteration
                        s_b2_done <= '1';
                    end if;
                end if;
            else
                v_b2_pointer := '0'; -- Initialization for starting next iteration (once blocks 3 finishes)
                s_b2_pointer <= '0';
                s_b2_wait <= '0';
                s_b2_up_CP <= '0';
                s_b2_done <= '0';
            end if;
        end if;
    end process;

    o_b2_P <= s_b2_P;
    o_b2_C <= s_b2_Co;
    o_b2_wait <= s_b2_wait;
    o_b2_up_CP <= s_b2_up_CP;
    o_b2_done <= s_b2_done;

end Behavioral;
```

9.2 Block 2 testbench

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity tb_Block_2 is
    generic (
        C_BLOCK_SIZE: integer := 256    -- Number of bits of the incoming message
    );
end tb_Block_2;

architecture Behavioral of tb_Block_2 is
    component Block_2
        port (
            i_b2_clk:   in std_logic;
            i_b2_reset: in std_logic;
            i_b2_en:    in std_logic;
            i_b2_charge:in std_logic;
            i_b2_M:     in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
            i_b2_E:     in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
            i_b2_P:     in std_logic_vector((C_BLOCK_SIZE-1) downto 0); -- Feedback
            from Block 3
            i_b2_C:     in std_logic_vector((C_BLOCK_SIZE-1) downto 0); -- Feedback
            from Block 3
            o_b2_P:     out std_logic_vector((C_BLOCK_SIZE-1) downto 0);
            o_b2_C:     out std_logic_vector((C_BLOCK_SIZE-1) downto 0);
            o_b2_wait:  out std_logic;
            o_b2_up_CP: out std_logic;
            o_b2_done:  out std_logic
        );
    end component;

    signal tb_b2_clk:   std_logic;
    signal tb_b2_reset: std_logic;
    signal tb_b2_en:    std_logic;
    signal tb_b2_charge:std_logic;
    signal tb_b2_M:     std_logic_vector((C_BLOCK_SIZE-1) downto 0);
    signal tb_b2_E:     std_logic_vector((C_BLOCK_SIZE-1) downto 0);
    signal tb_b2_P:     std_logic_vector((C_BLOCK_SIZE-1) downto 0);
    signal tb_b2_C:     std_logic_vector((C_BLOCK_SIZE-1) downto 0);
    signal tbo_b2_P:    std_logic_vector((C_BLOCK_SIZE-1) downto 0);
    signal tbo_b2_C:    std_logic_vector((C_BLOCK_SIZE-1) downto 0);
    signal tb_b2_wait:  std_logic;
    signal tb_b2_up_CP: std_logic;
    signal tb_b2_done:  std_logic;

begin
    uut: Block_2 port map (
        i_b2_clk => tb_b2_clk,
        i_b2_reset => tb_b2_reset,
        i_b2_en => tb_b2_en,
        i_b2_charge => tb_b2_charge,
        i_b2_M => tb_b2_M,
        i_b2_E => tb_b2_E,
        i_b2_P => tb_b2_P,
        i_b2_C => tb_b2_C,
        o_b2_P => tbo_b2_P,
        o_b2_C => tbo_b2_C,
        o_b2_wait => tb_b2_wait,
        o_b2_up_CP => tb_b2_up_CP,
        o_b2_done => tb_b2_done
    );

    tb0 : process
    begin
        tb_b2_clk <= '1'; wait for 2.5 ns;
        tb_b2_clk <= '0'; wait for 2.5 ns;
    end process;

    tb1 : process
    begin
        tb_b2_M <= "00000000000000000000000000000000";
        tb_b2_E <= "00000000000000000000000000000000";
        tb_b2_P <= "00000000000000000000000000000000";
        tb_b2_C <= "00000000000000000000000000000000";
        tb_b2_charge <= '0';
        tb_b2_en <= '0';
        tb_b2_reset <= '0'; wait for 5 ns;
        tb_b2_reset <= '1'; wait for 5 ns;

        tb_b2_M <= "00110011001100110011001100110011";
        tb_b2_E <= "01010010100110100101100101101010";
        tb_b2_P <= "01100101001100110010111000101001";
        tb_b2_C <= "00001111000011110000111100001111";
        tb_b2_charge <= '1';
        tb_b2_en <= '1'; wait for 10 ns;
        tb_b2_charge <= '0';
        tb_b2_en <= '0'; wait for 10 ns;
        tb_b2_en <= '1'; wait for 10 ns;
        tb_b2_en <= '0'; wait for 10 ns;
        tb_b2_en <= '1'; wait for 10 ns;
        tb_b2_en <= '0'; wait for 10 ns;
        tb_b2_en <= '1'; wait for 10 ns;
        tb_b2_en <= '0'; wait for 10 ns;
        tb_b2_en <= '1'; wait for 10 ns;
        tb_b2_en <= '0'; wait for 10 ns;
        tb_b2_en <= '1'; wait for 10 ns;
        tb_b2_en <= '0'; wait for 10 ns;
        tb_b2_en <= '1'; wait for 10 ns;
        tb_b2_en <= '0'; wait for 10 ns;
        tb_b2_en <= '1'; wait for 10 ns;
        tb_b2_en <= '0'; wait for 10 ns;

        tb_b2_P <= "11010101010101001101101110101010";
        tb_b2_C <= "11110000111100001111000011110000";

        tb_b2_charge <= '1';
        tb_b2_en <= '1'; wait for 10 ns;
        tb_b2_charge <= '0';
        tb_b2_en <= '0'; wait for 10 ns;
        tb_b2_en <= '1'; wait for 10 ns;
        tb_b2_en <= '0'; wait for 10 ns;
        tb_b2_en <= '1'; wait for 10 ns;
        tb_b2_en <= '0'; wait for 10 ns;
        tb_b2_en <= '1'; wait for 10 ns;
        tb_b2_en <= '0'; wait for 10 ns;
        tb_b2_en <= '1'; wait for 10 ns;
        tb_b2_en <= '0'; wait for 10 ns;
        tb_b2_en <= '1'; wait for 10 ns;
        tb_b2_en <= '0'; wait for 10 ns;
        tb_b2_en <= '1'; wait for 10 ns;
        tb_b2_en <= '0'; wait for 10 ns;
        tb_b2_en <= '1'; wait for 10 ns;
        tb_b2_en <= '0'; wait for 50 ns;    -- Simulate with 400 ns
    end process;

end Behavioral;
```

9.3 Block 3 code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
USE ieee.numeric_std.ALL;   -- For function "To_integer"

entity Block_3 is
    generic (
        C_BLOCK_SIZE: integer := 256    -- Number of bits of the incoming message
    );
    port (
        i_b3_clk:   in std_logic;
        i_b3_reset: in std_logic;
        i_b3_en:    in std_logic;
        i_b3_A:     in std_logic_vector(C_BLOCK_SIZE-1 downto 0);
        i_b3_B:     in std_logic_vector(C_BLOCK_SIZE-1 downto 0);
        i_b3_N:     in std_logic_vector(C_BLOCK_SIZE-1 downto 0);
        o_b3_C:     out std_logic_vector(C_BLOCK_SIZE-1 downto 0);
        o_b3_done:  out std_logic
    );
end Block_3;

architecture Behavioral of Block_3 is
    signal s_b3_C:     std_logic_vector (C_BLOCK_SIZE-1 downto 0);
    signal s_b3_done:  std_logic;
    signal s_b3_index: integer range 0 to C_BLOCK_SIZE;
begin
    process (i_b3_clk, i_b3_reset) is
    variable v_b3_C: std_logic_vector ((C_BLOCK_SIZE+2) downto 0);  -- Extra length to ensure the multiplication is used for will fit
    variable v_b3_pointer: std_logic;   -- Elements updated and then checked in the same process MUST be variable, not signal,
    variable v_start: std_logic;        -- because signals are just updated once the process finishes!!!!!
    begin
        if (i_b3_reset = '0') then
            s_b3_C <= (others => '0');
            v_b3_C := (others => '0');
            v_b3_pointer := '0';
            s_b3_done <= '0';
            s_b3_index <= 0;
            v_start := '0';
        elsif (rising_edge(i_b3_clk)) then
            if (i_b3_en = '1') then
                if (s_b3_index <= (C_BLOCK_SIZE-1)) then
                    if ((s_b3_index = 0) and (i_b3_B(C_BLOCK_SIZE-1) = '0')) then -- Only the first time (and if first bit to check is 0)
                        v_start := '0';
                        for k in 0 to (C_BLOCK_SIZE-1) loop
                            v_start := v_start or i_b3_B((C_BLOCK_SIZE-1)-k);   -- Multiplication algorithm is done from Left to Right!
                            if (v_start = '1') then
                                s_b3_index <= k;
                                exit;   -- It leaves the for loop
                            end if;
                        end loop;
                    else
                        v_b3_pointer := i_b3_B((C_BLOCK_SIZE-1)-s_b3_index);    -- Multiplication algorithm is done from Left to Right!
                        if (v_b3_pointer = '1') then
                            v_b3_C := std_logic_vector(shift_left(unsigned(v_b3_C), 1) + unsigned(i_b3_A));
                        else
                            v_b3_C := std_logic_vector(shift_left(unsigned(v_b3_C), 1));
                        end if;
                        if (v_b3_C > i_b3_N) then           -- Up to 2 subtracts, if the number is big enough
                            v_b3_C := v_b3_C - i_b3_N;
                        end if;
                        if (v_b3_C > i_b3_N) then   -- If a value must be updated and checked in the same process, it must be a variable!
                            v_b3_C := v_b3_C - i_b3_N;
                        end if;
                        s_b3_C <= v_b3_C ((C_BLOCK_SIZE-1) downto 0);       -- Output is updated continuously
                        s_b3_index <= s_b3_index + 1;
                    end if;
                else
                    s_b3_C <= v_b3_C ((C_BLOCK_SIZE-1) downto 0);
                    s_b3_done <= '1';
                end if;
            else
                v_b3_pointer := '0';        -- Data to restart once the block is disabled
                v_b3_C := (others => '0');  -- This variable must restart every time the block restarts to ensure the MSB 0s are ignored
                s_b3_done <= '0';
                s_b3_index <= 0;
            end if;
        end if;
    end process;

    o_b3_C <= s_b3_C;
    o_b3_done <= s_b3_done;

end Behavioral;
```

9.4 Block 3 testbench

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity tb_Block_3 is
    generic (
        C_BLOCK_SIZE: integer := 256    -- Number of bits of the incoming message
    );
end tb_Block_3;

architecture Behavioral of tb_Block_3 is
    component Block_3
        port (
            i_b3_clk:   in std_logic;
            i_b3_reset: in std_logic;
            i_b3_en:    in std_logic;
            i_b3_A:     in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
            i_b3_B:     in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
            i_b3_N:     in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
            o_b3_C:     out std_logic_vector((C_BLOCK_SIZE-1) downto 0);
            o_b3_done:  out std_logic
        );
    end component;

    signal tb_b3_clk:   std_logic;
    signal tb_b3_reset: std_logic;
    signal tb_b3_en:    std_logic;
    signal tb_b3_A:     std_logic_vector((C_BLOCK_SIZE-1) downto 0);
    signal tb_b3_B:     std_logic_vector((C_BLOCK_SIZE-1) downto 0);
    signal tb_b3_N:     std_logic_vector((C_BLOCK_SIZE-1) downto 0);
    signal tb_b3_C:     std_logic_vector((C_BLOCK_SIZE-1) downto 0);
    signal tb_b3_done:  std_logic;

begin
    uut: Block_3 port map (
        i_b3_clk => tb_b3_clk,
        i_b3_reset => tb_b3_reset,
        i_b3_en => tb_b3_en,
        i_b3_A => tb_b3_A,
        i_b3_B => tb_b3_B,
        i_b3_N => tb_b3_N,
        o_b3_C => tb_b3_C,
        o_b3_done => tb_b3_done
    );

    tb0 : process
    begin
        tb_b3_clk <= '1'; wait for 2 ns;
        tb_b3_clk <= '0'; wait for 2 ns;
    end process;

    tb1 : process
    begin
    -- These numbers are coprimes, and E (B) must be smaller than N (https://www.dcode.fr/coprimes)
        tb_b3_A <= "10100110110000111110011101010101";  -- 2797856597
        tb_b3_B <= "10100111011001000111001110111010";  -- 2808378298
        tb_b3_N <= "11111000011001100100111100001011";  -- 4167454475
        -- tb_b3_A <= "00000000011110100001101111010001";   -- 8002513
        -- tb_b3_B <= "00000000101011001110011101101101";   -- 11331437
        -- tb_b3_N <= "00000000111001100100111100001011";   -- 15093515
        tb_b3_en <= '0';
        tb_b3_reset <= '0'; wait for 10 ns;
        tb_b3_reset <= '1'; wait for 10 ns;
        tb_b3_en <= '1'; wait for 150 ns;
        tb_b3_en <= '0'; wait for 10 ns;    -- Simulate with 200 ns
    end process;

end Behavioral;
```

9.5 Datapath code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
USE ieee.numeric_std.ALL;   -- For function "To_integer"

entity Datapath is
    generic (
        C_BLOCK_SIZE: integer := 256    -- Number of bits of the incoming message
    );
    port (
        i_data_clk:         in std_logic;
        i_data_rst:         in std_logic;
        i_data_M:           in std_logic_vector(C_BLOCK_SIZE-1 downto 0);
        i_data_E:           in std_logic_vector(C_BLOCK_SIZE-1 downto 0);
        i_data_N:           in std_logic_vector(C_BLOCK_SIZE-1 downto 0);
        o_data_C:           out std_logic_vector(C_BLOCK_SIZE-1 downto 0);

        i_data_b2_en:       in std_logic;
        i_data_b2_charge:   in std_logic;
        o_data_b2_wait:     out std_logic;
        o_data_b2_up_CP:    out std_logic;
        o_data_b2_done:     out std_logic;

        i_data_b31_en:      in std_logic;
        i_data_b32_en:      in std_logic;
        o_data_b31_done:    out std_logic;
        o_data_b32_done:    out std_logic
    );
end Datapath;

architecture Behavioral of Datapath is
    component Block_2 is
        port (
            i_b2_clk:   in std_logic;
            i_b2_reset: in std_logic;
            i_b2_en:    in std_logic;
            i_b2_charge:in std_logic;
            i_b2_M:     in std_logic_vector(C_BLOCK_SIZE-1 downto 0);
            i_b2_E:     in std_logic_vector(C_BLOCK_SIZE-1 downto 0);
            i_b2_P:     in std_logic_vector(C_BLOCK_SIZE-1 downto 0);   -- Feedback from Block 3
            i_b2_C:     in std_logic_vector(C_BLOCK_SIZE-1 downto 0);   -- Feedback from Block 3
            o_b2_P:     out std_logic_vector(C_BLOCK_SIZE-1 downto 0);
            o_b2_C:     out std_logic_vector(C_BLOCK_SIZE-1 downto 0);
            o_b2_wait:  out std_logic;
            o_b2_up_CP: out std_logic;
            o_b2_done:  out std_logic
        );
    end component Block_2;

    component Block_3 is
        port (
            i_b3_clk:   in std_logic;
            i_b3_reset: in std_logic;
            i_b3_en:    in std_logic;
            i_b3_A:     in std_logic_vector(C_BLOCK_SIZE-1 downto 0);
            i_b3_B:     in std_logic_vector(C_BLOCK_SIZE-1 downto 0);
            i_b3_N:     in std_logic_vector(C_BLOCK_SIZE-1 downto 0);
            o_b3_C:     out std_logic_vector(C_BLOCK_SIZE-1 downto 0);
            o_b3_done:  out std_logic
        );
    end component Block_3;

    signal s_A: std_logic_vector (C_BLOCK_SIZE-1 downto 0);
    signal s_B: std_logic_vector (C_BLOCK_SIZE-1 downto 0);
    signal s_P: std_logic_vector (C_BLOCK_SIZE-1 downto 0);
    signal s_C: std_logic_vector (C_BLOCK_SIZE-1 downto 0);

```

```vhdl
begin
    b1: Block_2 port map (      -- Main block (for loop)
        i_b2_clk => i_data_clk,
        i_b2_reset => i_data_rst,
        i_b2_en => i_data_b2_en,
        i_b2_charge => i_data_b2_charge,
        i_b2_M => i_data_M,
        i_b2_E => i_data_E,
        i_b2_P => s_P,
        i_b2_C => s_C,
        o_b2_P => s_B,
        o_b2_C => s_A,
        o_b2_wait => o_data_b2_wait,
        o_b2_up_CP => o_data_b2_up_CP,
        o_b2_done => o_data_b2_done
    );
    b2: Block_3 port map (      -- Secondary block (for loop), in parallel with the other block 3
        i_b3_clk => i_data_clk,
        i_b3_reset => i_data_rst,
        i_b3_en => i_data_b31_en,
        i_b3_A => s_A,      -- C = C * P mod N
        i_b3_B => s_B,
        i_b3_N => i_data_N,
        o_b3_C => s_C,
        o_b3_done => o_data_b31_done
    );
    b3: Block_3 port map (      -- Secondary block (for loop), in parallel with the other block 3
        i_b3_clk => i_data_clk,
        i_b3_reset => i_data_rst,
        i_b3_en => i_data_b32_en,
        i_b3_A => s_B,          -- P = P * P mod N
        i_b3_B => s_B,
        i_b3_N => i_data_N,
        o_b3_C => s_P,
        o_b3_done => o_data_b32_done
    );
    o_data_C <= s_A;

end Behavioral;
```

9.6 Datapath testbench

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity tb_Datapath is
    generic (
        C_BLOCK_SIZE: integer := 256    -- Number of bits of the incoming message
    );
end tb_Datapath;

architecture Behavioral of tb_Datapath is
    component Datapath
        port (
            i_data_clk:         in std_logic;
            i_data_rst:         in std_logic;
            i_data_M:           in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
            i_data_E:           in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
            i_data_N:           in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
            o_data_C:           out std_logic_vector((C_BLOCK_SIZE-1) downto 0);

            i_data_b2_en:       in std_logic;
            i_data_b2_charge:   in std_logic;
            o_data_b2_wait:     out std_logic;
            o_data_b2_up_CP:    out std_logic;
            o_data_b2_done:     out std_logic;

            i_data_b31_en:      in std_logic;
            i_data_b32_en:      in std_logic;
            o_data_b31_done:    out std_logic;
            o_data_b32_done:    out std_logic
        );
    end component;

    signal tb_data_clk: std_logic;
    signal tb_data_rst: std_logic;
    signal tb_data_E:   std_logic_vector((C_BLOCK_SIZE-1) downto 0);
    signal tb_data_N:   std_logic_vector((C_BLOCK_SIZE-1) downto 0);
    signal tb_data_M:   std_logic_vector((C_BLOCK_SIZE-1) downto 0);
    signal tb_data_C:   std_logic_vector((C_BLOCK_SIZE-1) downto 0);

    signal tb_data_b2_en:     std_logic;
    signal tb_data_b2_charge:std_logic;
    signal tb_data_b2_wait:   std_logic;
    signal tb_data_b2_up_CP: std_logic;
    signal tb_data_b2_done:   std_logic;

    signal tb_data_b31_en:    std_logic;
    signal tb_data_b32_en:    std_logic;
    signal tb_data_b31_done: std_logic;
    signal tb_data_b32_done: std_logic;

begin
    uut: Datapath port map (
        i_data_clk => tb_data_clk,
        i_data_rst => tb_data_rst,
        i_data_E => tb_data_E,
        i_data_N => tb_data_N,
        i_data_M => tb_data_M,
        o_data_C => tb_data_C,

        i_data_b2_en => tb_data_b2_en,
        i_data_b2_charge => tb_data_b2_charge,
        o_data_b2_wait => tb_data_b2_wait,
        o_data_b2_up_CP => tb_data_b2_up_CP,
        o_data_b2_done => tb_data_b2_done,

        i_data_b31_en => tb_data_b31_en,
        i_data_b32_en => tb_data_b32_en,
        o_data_b31_done => tb_data_b31_done,
        o_data_b32_done => tb_data_b32_done
    );

    tb0 : process
    begin
        tb_data_clk <= '1'; wait for 2 ns;
        tb_data_clk <= '0'; wait for 2 ns;
    end process;

    tb1 : process
    begin
        tb_data_b2_en <= '0';
        tb_data_b2_charge <= '0';
        tb_data_b31_en <= '0';
        tb_data_b32_en <= '0';
        tb_data_M <= "00000000101110100001101111010001";
        tb_data_E <= "00000000101011001110011101101101";
        tb_data_N <= "00000000111001100100111100001011";
        tb_data_rst <= '0';     wait for 10 ns;
        tb_data_rst <= '1';     wait for 10 ns;

        tb_data_b2_en <= '1';
        tb_data_b2_charge <= '1';   wait for 5 ns;
        tb_data_b2_charge <= '0';
        tb_data_b2_en <= '0';   wait for 10 ns;
        tb_data_b31_en <= '1';
        tb_data_b32_en <= '1';  wait for 135 ns;
        tb_data_b31_en <= '0';
        tb_data_b32_en <= '0';  wait for 10 ns;

        tb_data_b2_en <= '1';   wait for 10 ns;
        tb_data_b2_en <= '0';   wait for 10 ns;
        tb_data_b32_en <= '1';  wait for 135 ns;
        tb_data_b32_en <= '0';  wait for 10 ns;

        tb_data_b2_en <= '1';   wait for 10 ns;
        tb_data_b2_en <= '0';   wait for 10 ns;
        tb_data_b31_en <= '1';
        tb_data_b32_en <= '1';  wait for 135 ns;
        tb_data_b31_en <= '0';
        tb_data_b32_en <= '0';  wait for 10 ns;

        tb_data_b2_en <= '1';   wait for 10 ns;
        tb_data_b2_en <= '0';   wait for 10 ns;
        tb_data_b32_en <= '1';  wait for 135 ns;
        tb_data_b32_en <= '0';  wait for 10 ns;

        tb_data_b2_en <= '1';   wait for 10 ns;
        tb_data_b2_en <= '0';   wait for 10 ns;
        tb_data_b31_en <= '1';
        tb_data_b32_en <= '1';  wait for 135 ns;
        tb_data_b31_en <= '0';
        tb_data_b32_en <= '0';  wait for 10 ns; -- Simulate with 890 ns
    end process;

end architecture Behavioral;
```

9.7 Controller code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
USE ieee.numeric_std.ALL;   -- For function "To_integer"

entity Controller is    -- FSM that manages the rest of the blocks
    port (
        i_FSM_clk:      in std_logic;
        i_FSM_reset:    in std_logic;

        msgin_valid:    in std_logic;
        msgin_ready:    out std_logic;
        msgin_last:     in std_logic;

        msgout_valid:   out std_logic;
        msgout_ready:   in std_logic;
        msgout_last:    out std_logic;

        i_b2_wait:      in std_logic;
        i_b2_up_CP:     in std_logic;
        i_b2_done:      in std_logic;
        o_b2_charge:    out std_logic;
        o_b2_en:        out std_logic;

        i_b31_done:     in std_logic;
        i_b32_done:     in std_logic;
        o_b31_en:       out std_logic;
        o_b32_en:       out std_logic;

        o_fsm_state:    out std_logic_vector(31 downto 0)
    );
end Controller;

architecture Behavioral of Controller is
    type STATES is (IDLE, CHARGE_VALUES, WORK_B2, WORK_B32, WORK_BOTH_B3, FINISHED);
    signal state_next, state_reg: STATES;
    signal s_b2_charge_next, s_b2_charge_reg: std_logic;
    signal s_b2_en_next, s_b2_en_reg: std_logic;
    signal s_b31_en_next, s_b31_en_reg: std_logic;
    signal s_b32_en_next, s_b32_en_reg: std_logic;

    signal msgin_ready_next, msgin_ready_reg: std_logic;
    signal msgout_valid_next, msgout_valid_reg: std_logic;
    signal msgout_last_next, msgout_last_reg: std_logic;

    signal s_fsm_state_next, s_fsm_state_reg: std_logic_vector (31 downto 0);

    signal s_msg_last_next, s_msg_last_reg: std_logic;
    signal s_msg_accept: std_logic;

begin
    -- FSM: state and data registers
    process (i_FSM_clk, i_FSM_reset)
    begin
        if (i_FSM_reset = '0') then
            state_reg <= IDLE;
            msgin_ready_reg <= '0';
            msgout_valid_reg <= '0';
            msgout_last_reg <= '0';
            s_msg_last_reg <= '0';
            s_fsm_state_reg <= (others => '0');
            s_b2_charge_reg <= '0';
            s_b2_en_reg <= '0';
            s_b31_en_reg <= '0';
            s_b32_en_reg <= '0';
        elsif (rising_edge(i_FSM_clk)) then
            state_reg <= state_next;
            msgin_ready_reg <= msgin_ready_next;
            msgout_valid_reg <= msgout_valid_next;
            msgout_last_reg <= msgout_last_next;
            if ((state_reg = IDLE) or (state_reg = CHARGE_VALUES)) then
                s_msg_last_reg <= s_msg_last_next;
            end if;
            s_fsm_state_reg <= s_fsm_state_next;
            s_b2_charge_reg <= s_b2_charge_next;
            s_b2_en_reg <= s_b2_en_next;
            s_b31_en_reg <= s_b31_en_next;
            s_b32_en_reg <= s_b32_en_next;
        end if;
    end process;

    -- Control flow
    process (msgin_valid, i_b2_wait, i_b2_done, i_b2_up_CP, i_b31_done, i_b32_done, state_reg, msgin_last, msgin_ready_reg, msgout_ready,
            msgout_valid_reg, msgout_last_reg, s_msg_last_reg, s_fsm_state_reg, s_b2_charge_reg, s_b2_en_reg, s_b31_en_reg, s_b32_en_reg, s_msg_accept)
    begin
        state_next <= state_reg;        -- NOTE: Here XX_reg cannot be updated. Otherwise there will be an error!
        msgin_ready_next <= '0';
        msgout_valid_next <= '0';
        msgout_last_next <= '0';
        s_msg_last_next <= '0';
        s_fsm_state_next <= (others => '0');
        s_b2_charge_next <= '0';
        s_b2_en_next <= '0';
        s_b31_en_next <= '0';
        s_b32_en_next <= '0';

        case state_reg is
            when IDLE =>
                s_fsm_state_next <= (others => '0');
                msgin_ready_next <= '0';
                msgout_valid_next <= '0';
                msgout_last_next <= '0';
                s_msg_last_next <= '0';
                s_b2_charge_next <= '0';
                s_b2_en_next <= '0';
                s_b31_en_next <= '0';
                s_b32_en_next <= '0';
                if (msgin_valid = '1') then
                    state_next <= CHARGE_VALUES;
                end if;
            when CHARGE_VALUES =>
                s_fsm_state_next(2 downto 0) <= "001";
                msgin_ready_next <= '1';
                s_msg_last_next <= msgin_last;
                s_b2_charge_next <= '1';
                s_b2_en_next <= '1';
                if (i_b2_wait = '0') then
                    state_next <= WORK_B2;
                end if;
            when WORK_B2 =>
                s_fsm_state_next(2 downto 0) <= "010";
                msgin_ready_next <= '0';
                s_b2_charge_next <= '0';
                s_b2_en_next <= '1';
                s_b31_en_next <= '0';
                s_b32_en_next <= '0';
                if (i_b2_done = '1') then
                    state_next <= FINISHED;
                elsif (i_b2_wait = '1') then
                    if (i_b2_up_CP = '1') then
                        state_next <= WORK_BOTH_B3;
                    else
                        state_next <= WORK_B32;
                    end if;
                end if;
            when WORK_B32 =>
                s_fsm_state_next(2 downto 0) <= "011";
                s_b2_en_next <= '0';
                s_b31_en_next <= '0';
                s_b32_en_next <= '1';
                if (i_b32_done = '1') then
                    state_next <= WORK_B2;
                end if;
            when WORK_BOTH_B3 =>
                s_fsm_state_next(2 downto 0) <= "100";
                s_b2_en_next <= '0';
                s_b31_en_next <= '1';
                s_b32_en_next <= '1';
                if ((i_b31_done = '1') and (i_b32_done = '1')) then
                    state_next <= WORK_B2;
                end if;
            when FINISHED =>
                s_fsm_state_next(2 downto 0) <= "101";
                msgout_valid_next <= '1';
                msgout_last_next <= s_msg_last_reg;
                s_b2_en_next <= '0';
                s_b31_en_next <= '0';
                s_b32_en_next <= '0';
                if (s_msg_accept = '1') then
                    msgout_valid_next <= '0';
                    state_next <= IDLE;
                end if;
        end case;
    end process;

    s_msg_accept <= msgout_ready and msgout_valid_reg;

    msgin_ready <= msgin_ready_reg;
    msgout_valid <= msgout_valid_reg;
    msgout_last <= msgout_last_reg;
    o_b2_charge <= s_b2_charge_reg;
    o_b2_en <= s_b2_en_reg;
    o_b31_en <= s_b31_en_reg;
    o_b32_en <= s_b32_en_reg;
    o_fsm_state <= s_fsm_state_reg;

end Behavioral;
```

9.8 Controller testbench

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity tb_Controller is
end tb_Controller;

architecture Behavioral of tb_Controller is
    component Controller
        port (
            i_FSM_clk:      in std_logic;
            i_FSM_reset:    in std_logic;
            msgin_valid:    in std_logic;
            msgin_ready:    out std_logic;
            msgin_last:     in std_logic;
            msgout_valid:   out std_logic;
            msgout_ready:   in std_logic;
            msgout_last:    out std_logic;
            i_b2_wait:      in std_logic;
            i_b2_up_CP:     in std_logic;
            i_b2_done:      in std_logic;
            o_b2_charge:    out std_logic;
            o_b2_en:        out std_logic;
            i_b31_done:     in std_logic;
            i_b32_done:     in std_logic;
            o_b31_en:       out std_logic;
            o_b32_en:       out std_logic;
            o_fsm_state:    out std_logic_vector(31 downto 0)
        );
    end component;

    signal tb_FSM_clk:      std_logic;
    signal tb_FSM_reset:    std_logic;
    signal tb_msgin_valid:  std_logic;
    signal tb_msgin_ready:  std_logic;
    signal tb_msgin_last:   std_logic;
    signal tb_msgout_valid: std_logic;
    signal tb_msgout_ready: std_logic;
    signal tb_msgout_last:  std_logic;
    signal tb_b2_wait:      std_logic;
    signal tb_b2_up_CP:     std_logic;
    signal tb_b2_done:      std_logic;
    signal tb_b2_charge:    std_logic;
    signal tb_b2_en:        std_logic;
    signal tb_b31_done:     std_logic;
    signal tb_b32_done:     std_logic;
    signal tb_b31_en:       std_logic;
    signal tb_b32_en:       std_logic;
    signal tb_fsm_state:    std_logic_vector(31 downto 0);

begin
    uut: Controller port map (
        i_FSM_clk => tb_FSM_clk,
        i_FSM_reset => tb_FSM_reset,
        msgin_valid => tb_msgin_valid,
        msgin_ready => tb_msgin_ready,
        msgin_last => tb_msgin_last,
        msgout_valid => tb_msgout_valid,
        msgout_ready => tb_msgout_ready,
        msgout_last => tb_msgout_last,
        i_b2_wait => tb_b2_wait,
        i_b2_up_CP => tb_b2_up_CP,
        i_b2_done => tb_b2_done,
        o_b2_charge => tb_b2_charge,
        o_b2_en => tb_b2_en,
        i_b31_done => tb_b31_done,
        i_b32_done => tb_b32_done,
        o_b31_en => tb_b31_en,
        o_b32_en => tb_b32_en,
        o_fsm_state => tb_fsm_state
    );

    tb0 : process
    begin
        tb_FSM_clk <= '1'; wait for 2 ns;
        tb_FSM_clk <= '0'; wait for 2 ns;
    end process;

    tb1 : process
    begin
        tb_msgin_valid <= '0';
        tb_msgin_last <= '0';
        tb_msgout_ready <= '0';
        tb_b2_wait <= '0';
        tb_b2_up_CP <= '0';
        tb_b2_done <= '0';
        tb_b31_done <= '0';
        tb_b32_done <= '0';
        tb_FSM_reset <= '0'; wait for 5 ns;
        tb_FSM_reset <= '1'; wait for 5 ns;

        tb_msgin_valid <= '1'; wait for 10 ns;
        tb_msgin_valid <= '0'; wait for 10 ns;
        tb_b2_wait <= '1';  wait for 10 ns;
        tb_b2_wait <= '0';  wait for 10 ns;
        tb_b32_done <= '1'; wait for 10 ns;
        tb_b31_done <= '0'; wait for 10 ns;
        tb_b32_done <= '0'; wait for 10 ns;

        tb_b2_up_CP <= '1';
        tb_b2_wait <= '1';  wait for 10 ns;
        tb_b2_up_CP <= '0';
        tb_b2_wait <= '0';  wait for 10 ns;
        tb_b31_done <= '1'; wait for 10 ns;
        tb_b32_done <= '1'; wait for 10 ns;
        tb_b31_done <= '0'; wait for 10 ns;
        tb_b32_done <= '0'; wait for 10 ns;

        tb_b2_done <= '1';  wait for 10 ns;
        tb_b2_done <= '0';  wait for 30 ns;

        tb_msgout_ready <= '1'; wait for 50 ns;
        tb_msgin_valid <= '1';
        tb_msgin_last <= '1'; wait for 10 ns;
        tb_msgin_valid <= '0';
        tb_msgin_last <= '0'; wait for 10 ns;
        tb_msgout_ready <= '1';
        tb_b2_wait <= '1';  wait for 10 ns;
        tb_b2_wait <= '0';  wait for 10 ns;
        tb_b32_done <= '1'; wait for 10 ns;
        tb_b31_done <= '0'; wait for 10 ns;
        tb_b32_done <= '0'; wait for 10 ns;

        tb_b2_up_CP <= '1';
        tb_b2_wait <= '1';  wait for 10 ns;
        tb_b2_up_CP <= '0';
        tb_b2_wait <= '0';  wait for 10 ns;
        tb_b31_done <= '1'; wait for 10 ns;
        tb_b32_done <= '1'; wait for 10 ns;
        tb_b31_done <= '0'; wait for 10 ns;
        tb_b32_done <= '0'; wait for 10 ns;

        tb_b2_done <= '1';  wait for 10 ns;
        tb_b2_done <= '0';  wait for 50 ns;
    end process;

end Behavioral;
```

9.9 RSA core code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
USE ieee.numeric_std.ALL;   -- For function "To_integer"

entity rsa_core is
    generic (
        C_BLOCK_SIZE: integer := 256    -- Number of bits of the incoming message
    );
    port (
        clk:            in std_logic;
        reset_n:        in std_logic;

        msgin_valid:    in std_logic;
        msgin_ready:    out std_logic;
        msgin_data:     in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
        msgin_last:     in std_logic;

        msgout_valid:   out std_logic;
        msgout_ready:   in std_logic;
        msgout_data:    out std_logic_vector((C_BLOCK_SIZE-1) downto 0);
        msgout_last:    out std_logic;

        key_e_d:        in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
        key_n:          in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
        rsa_status:     out std_logic_vector(31 downto 0)
    );
end rsa_core;

architecture Behavioral of rsa_core is
    component Datapath is
        port (
            i_data_clk:         in std_logic;
            i_data_rst:         in std_logic;
            i_data_M:           in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
            i_data_E:           in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
            i_data_N:           in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
            o_data_C:           out std_logic_vector((C_BLOCK_SIZE-1) downto 0);

            i_data_b2_en:       in std_logic;
            i_data_b2_charge:   in std_logic;
            o_data_b2_wait:     out std_logic;
            o_data_b2_up_CP:    out std_logic;
            o_data_b2_done:     out std_logic;

            i_data_b31_en:      in std_logic;
            i_data_b32_en:      in std_logic;
            o_data_b31_done:    out std_logic;
            o_data_b32_done:    out std_logic
        );
    end component Datapath;

    component Controller is
        port (
            i_FSM_clk:      in std_logic;
            i_FSM_reset:    in std_logic;

            msgin_valid:    in std_logic;
            msgin_ready:    out std_logic;
            msgin_last:     in std_logic;

            msgout_valid:   out std_logic;
            msgout_ready:   in std_logic;
            msgout_last:    out std_logic;

            i_b2_wait:      in std_logic;
            i_b2_up_CP:     in std_logic;
            i_b2_done:      in std_logic;
            o_b2_charge:    out std_logic;
```

```vhdl
                o_b2_en:        out std_logic;

                i_b31_done:     in std_logic;
                i_b32_done:     in std_logic;
                o_b31_en:       out std_logic;
                o_b32_en:       out std_logic;

                o_fsm_state:    out std_logic_vector(31 downto 0)
            );
        end component Controller;

        signal s_b2_en: std_logic;
        signal s_b31_en: std_logic;
        signal s_b32_en: std_logic;
        signal s_b2_charge: std_logic;
        signal s_b2_wait: std_logic;
        signal s_b2_up_CP: std_logic;
        signal s_b2_done: std_logic;
        signal s_b31_done: std_logic;
        signal s_b32_done: std_logic;

    begin
        DP: Datapath port map (     -- Datapath for doing all necessary calculations
            i_data_clk => clk,
            i_data_rst => reset_n,
            i_data_M => msgin_data,
            i_data_E => key_e_d,
            i_data_N => key_n,
            o_data_C => msgout_data,

            i_data_b2_en => s_b2_en,
            i_data_b2_charge => s_b2_charge,
            o_data_b2_wait => s_b2_wait,
            o_data_b2_up_CP => s_b2_up_CP,
            o_data_b2_done => s_b2_done,

            i_data_b31_en => s_b31_en,
            i_data_b32_en => s_b32_en,
            o_data_b31_done => s_b31_done,
            o_data_b32_done => s_b32_done
        );
        FSM: Controller port map (  -- Controller that manages the datapath
            i_FSM_clk => clk,
            i_FSM_reset => reset_n,

            msgin_valid => msgin_valid,
            msgin_ready => msgin_ready,
            msgin_last => msgin_last,

            msgout_valid => msgout_valid,
            msgout_ready => msgout_ready,
            msgout_last => msgout_last,

            i_b2_wait => s_b2_wait,
            i_b2_up_CP => s_b2_up_CP,
            i_b2_done => s_b2_done,
            o_b2_charge => s_b2_charge,
            o_b2_en => s_b2_en,

            i_b31_done => s_b31_done,
            i_b32_done => s_b32_done,
            o_b31_en => s_b31_en,
            o_b32_en => s_b32_en,
            o_fsm_state => rsa_status
        );

    end Behavioral;
```

9.10 RSA core testbench

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity tb_rsa_core is
    generic (
        C_BLOCK_SIZE: integer := 256    -- Number of bits of the incoming message
    );
end tb_rsa_core;

architecture Behavioral of tb_rsa_core is
    component rsa_core
        port (
            clk:            in std_logic;
            reset_n:        in std_logic;

            msgin_valid:    in std_logic;
            msgin_ready:    out std_logic;
            msgin_data:     in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
            msgin_last:     in std_logic;

            msgout_valid:   out std_logic;
            msgout_ready:   in std_logic;
            msgout_data:    out std_logic_vector((C_BLOCK_SIZE-1) downto 0);
            msgout_last:    out std_logic;

            key_e_d:        in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
            key_n:          in std_logic_vector((C_BLOCK_SIZE-1) downto 0);
            rsa_status:     out std_logic_vector(31 downto 0)
        );
    end component;

    signal tb_clk:          std_logic;
    signal tb_reset_n:      std_logic;

    signal tb_msgin_valid:  std_logic;
    signal tb_msgin_ready:  std_logic;
    signal tb_msgin_data:   std_logic_vector((C_BLOCK_SIZE-1) downto 0);
    signal tb_msgin_last:   std_logic;

    signal tb_msgout_valid: std_logic;
    signal tb_msgout_ready: std_logic;
    signal tb_msgout_data:  std_logic_vector((C_BLOCK_SIZE-1) downto 0);
    signal tb_msgout_last:  std_logic;

    signal tb_key_e_d:      std_logic_vector((C_BLOCK_SIZE-1) downto 0);
    signal tb_key_n:        std_logic_vector((C_BLOCK_SIZE-1) downto 0);
    signal tb_rsa_status:   std_logic_vector(31 downto 0);

begin
    uut: rsa_core port map (
        clk => tb_clk,
        reset_n => tb_reset_n,

        msgin_valid => tb_msgin_valid,
        msgin_ready => tb_msgin_ready,
        msgin_data => tb_msgin_data,
        msgin_last => tb_msgin_last,

        msgout_valid => tb_msgout_valid,
        msgout_ready => tb_msgout_ready,
        msgout_data => tb_msgout_data,
        msgout_last => tb_msgout_last,

        key_e_d => tb_key_e_d,
        key_n => tb_key_n,
        rsa_status => tb_rsa_status
    );

    tb0 : process
    begin
        tb_clk <= '1'; wait for 2 ns;
        tb_clk <= '0'; wait for 2 ns;
    end process;

    tb1 : process
    begin
        tb_msgin_valid <= '0';
        tb_msgin_last <= '0';
        tb_msgout_ready <= '0';
        -- These numbers must be coprimes each other, and E must be smaller than N & M (https://www.dcode.fr/coprimes)
        -- tb_msgin_data <= "10111010000110111101000101100101";
        -- tb_key_e_d <= "10100111011001000111001110111000";
        -- tb_key_n <= "11111000011001100100111100001011";  -- 4167454475
        tb_msgin_data <= "00000000011110100001101111010001";    -- 8002513
        tb_key_e_d <= "00000000101011001110011101101101";    -- 11331437
        tb_key_n <= "00000000111001100100111100001011"; -- 15093515
        tb_reset_n <= '0';      wait for 10 ns;
        tb_reset_n <= '1';      wait for 10 ns;

        tb_msgin_valid <= '1';  wait for 10 ns;
        tb_msgin_valid <= '0';  wait for 5400 ns;
        tb_msgout_ready <= '1'; wait for 10 ns;
        tb_msgout_ready <= '0'; wait for 100 ns;

        tb_msgin_data <= "10100110110000111110011101010101";    -- 2797856597
        tb_key_e_d <= "10100111011001000111001110111010";    -- 2808378298
        tb_key_n <= "11111000011001100100111100001011"; -- 4167454475
        -- tb_msgin_data <= "00000000011110100001101111010001"; -- 8002513
        -- tb_key_e_d <= "00000000101011001110011101101101";    -- 11331437
        -- tb_key_n <= "00000000111001100100111100001011";  -- 15093515
        tb_msgin_valid <= '1';
        tb_msgin_last <= '1';   wait for 15 ns;     -- +5 because of waiting for msgin_ready
        tb_msgin_valid <= '0';
        tb_msgin_last <= '0';   wait for 5400 ns;
        tb_msgout_ready <= '1'; wait for 10 ns;
        tb_msgout_ready <= '0'; wait for 300 ns;
    end process;

end architecture Behavioral;
```

Bibliography

[1] Wikipedia. Electronic design automation. `https://en.wikipedia.org/wiki/Electronic_design_automation`.

[2] Wikipedia. Rsa cryptosystem. `https://en.wikipedia.org/wiki/RSA_(cryptosystem)`.

[3] Inc. Xilinx. Axi reference guide. `https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf`.