UNIVERSITÀ DI PISA

M.Sc. Cybersecurity

Hardware and Embedded Security Course

Rotary Substitution Table Cypher

(Encryption module)

Students

Francesco Bocchi

Luca Canuzzi

Davide Cossari

Professor

Sergio Saponara

Referent

Ing. Luca Crocetti

Summary

1. Specification Analysis 2
2. Block Diagram and design choices (RTL design module) 3
3. Expected waveforms 7
4. Testbench 10
5. Implementation of RTL design on FPGA and results 11
6. Static Timing Analysis (STA) 12

Table of Figures

Figure 1. Rotary Substitution Table (RST) at first round. 2
Figure 2. High-level schematic of RST. 4
Figure 3. Architecture of the check module of RST. 5
Figure 4. Architecture of the rot_table module. 5
Figure 5. Architecture of the substitution_law module. 6
Figure 6. Initial part of the TEST_WORKFLOW waveform. 7
Figure 7. HELLO_EXAMPLE waveform. 7
Figure 8. Waveform that shows the right behavior of signal ctxt_ready and ptxt_valid. 8
Figure 9. Target waveforms concerning ctxt_ready and ptxt_valid signals. 8
Figure 10. Waveform when key is not valid. 9
Figure 11. Waveform when ptxt_valid is 0 and key is installed. 9
Figure 12. Summary of resource usage after Fitting. 11
Figure 13. Comparison of Fmax during slow test at 85C and 0C. 12
Figure 14. Content of .sdc file. 12

1. Specification Analysis

The scope of the project is to design and implement a module that takes text characters and digits in input and encrypts them by returning a 16-bit ciphertext using the following rotary substitution table:

	$S[1]$	$S[11]$	$S[3]$	$S[9]$	$S[5]$	$S[7]$
$S[0]$	a/A	b/B	c/C	d/D	e/E	f/F
$S[10]$	g/G	h/H	i/I	j/J	k/K	l/L
$S[2]$	m/M	n/N	o/O	p/P	q/Q	r/R
$S[8]$	s/S	t/T	u/U	v/V	w/W	x/X
$S[4]$	y/Y	z/Z	0	1	2	3
$S[6]$	4	5	6	7	8	9

Figure 1. Rotary Substitution Table (RST) at first round.

Being *S*[0], …, *S*[11] the 12 characters of the substitution word *S*, each letter of the alphabet (case insensitive) and the digits (0 to 9) are substituted with the corresponding pair of *S* characters in the order row-column. For instance, the letter "a" (or "A") is substitued with the pair *S*[0] *S*[1], while digit 3 is substitued with the pair *S*[4] *S*[7].

The substitution word S is first initialized with the corresponding characters of key K taken in input by following the assignment S[0] = K[0], …, S[11] = K[11], then the first substitution is done according to RST in Fig. 1.

At the end of the plaintext encryption, if no errors arise, the S characters are circularly shifted on the right and on the bottom.

The encryption module has the following ports:

- **(input) clk**: clock signal.
- **(input) rst_n**: asynchronous active-low reset signal. The asynchronous property of this port permits it to trigger this signal without waiting for the next clock cycle.
- **(input) ptxt_valid**: as required by specification, this is a signal that must be asserted when the plaintext is provided in input. If its value is 1, the corresponding input data is to be considered valid and stable, otherwise if its value is 0 the plaintext is inconsistent.
- **(input) key**: key used in order to initialize the rotary substitution table for the encryption function.
- **(input) ptxt_char**: 1-byte plaintext character to be encrypted.
- **(output) ctxt_str**: ciphertext computed by the encryption function.
- **(output) ctxt_ready**: as required by specification, this is a signal that must be asserted when the ciphertext is available on the corresponding output port. If its value is 1, the output data is valid and stable, otherwise if its value is 0 the ciphertext is inconsistent.
- **(output) err_invalid_ptxt_char**: a flag to detect invalid plaintext characters (i.e characters that are not digits or letters).
- **(output) err_invalid_key**: a flag to detect invalid characters in the key (i.e., repeated characters or invalid characters).
- **(output) key_not_installed**: a flag to detect when the key is not yet installed.

2. Block Diagram and design choices (RTL design module)

In this section it is explained the design architecture and choices of the proposed solution.

The encryption module consists of a top-level module, called *rst_cipher,* and three sub-modules:

- *check_key*: this module implements combinatory logic that checks if the key contains any invalid/repeated characters.
- *rot_table*: this module implements sequential logic to store the RST and rotate it.
- *substitution_law*: this module performs the substitution of the plaintext provided in input and generates the correspondent ciphertext.

2.1. Design choices

The *rst_cipher* module takes in input a key for the initialization of the rotary substitution table. Since the key is a 12 characters string, it is stored in a one-dimensional packed array of 12 bytes.

Since the *rot_table* module stores and rotates the RST, it takes in input three main signals:

1. *key*: a 12-bytes array, representing the key to be installed.
2. *key_valid*: a 1-bit signal that represents the output of the combinatory network *check_key*. If this flag is equals to logic value 1, this means that the key is valid (i.e., key does not contain repeated or invalid chars) and the table is initialized with the default configuration shown in Figure 1, consequently the flag *is_table_initialized* is set to 1.
3. *ctxt_valid*: a 1-bit signal that represents the output of the *substitution_law* module. When the plaintext is encrypted without errors, the *ctxt_valid* flag is set to 1 and, by using the flag as a feedback wire to the *rot_table* module, the RST is correctly rotated.

Every plaintext character is processed in a single clock cycle by the combinatory network *substitution_law*. This module takes in input the following signals:

1. *ptxt_char*: an 8-bit signal that represents the plaintext character to be encrypted.
2. *ptxt_valid*: a 1-bit signal that is asserted before the encryption.
3. *rot_table*: a 12-byte array, that is the output of the *rot_table* module.
4. *is_key_installed*: a 1-bit signal used to know if the RST taken in input is valid. If this flag is equal to 0, then the plaintext will not be encrypted since the table is not yet ready.

The outputs of the substitution module consist of:

1. *ctxt_str*: a 16-bit signal that represents the ciphertext.
2. *ctxt_valid*: a 1-bit signal that indicates if the ciphertext is valid.
3. *err_invalid_ptxt_char*: a 1-bit signal that is set when the plaintext contains invalid characters.

Finally, the outputs are not directly sent to the pins but are stored by the main module *rst_cipher* in specific registers to avoid long combinatory output paths. So, the stored outputs are:

- *ctxt_str.* It represents the ciphertext.
- *ctxt_valid.* When this flag is equals to 1, the ciphertext on the output port is consistent.

- *err_invalid_ptxt_char.* This flag is equals to 1 when the plaintext is not valid (i.e., not letter or digits), otherwise is equals to 0.
- *err_invalid_key.* This flag is equals to 1 when the key is not valid, otherwise is equals to 0.
- *key_not_installed.* This flag is equals to 1 when the key is not installed, otherwise is equals to 0.

2.2. Block Diagrams

The following figure shows the overall architecture of the RST Encryption module. From the left to the right, they are represented:

- the inputs of the module.
- the *check_key*, *rot_table* and *substitution_law* modules.
- the output registers.

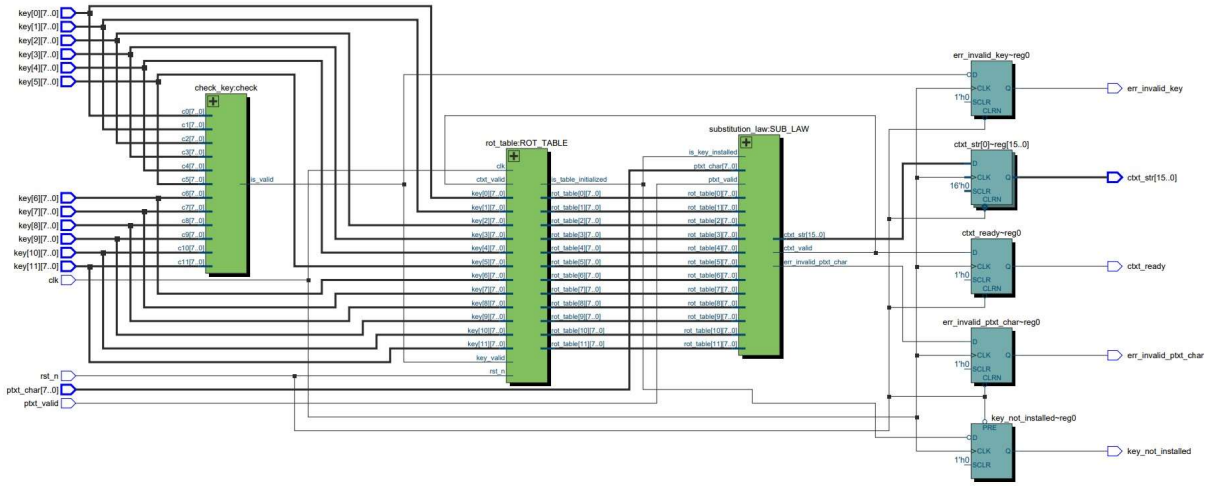

Figure 2. High-level schematic of RST.

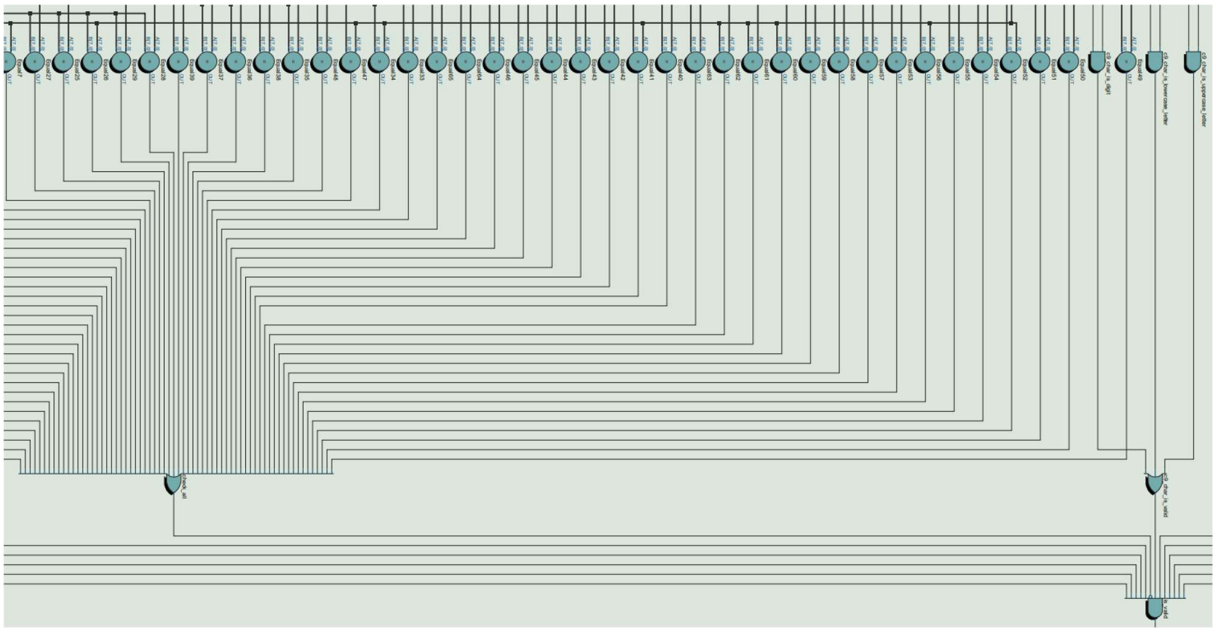

Figure 3. Architecture of the check module of RST.

Figure 3 represents a zoom of the *check_key* module, which verifies if the key is valid or not.

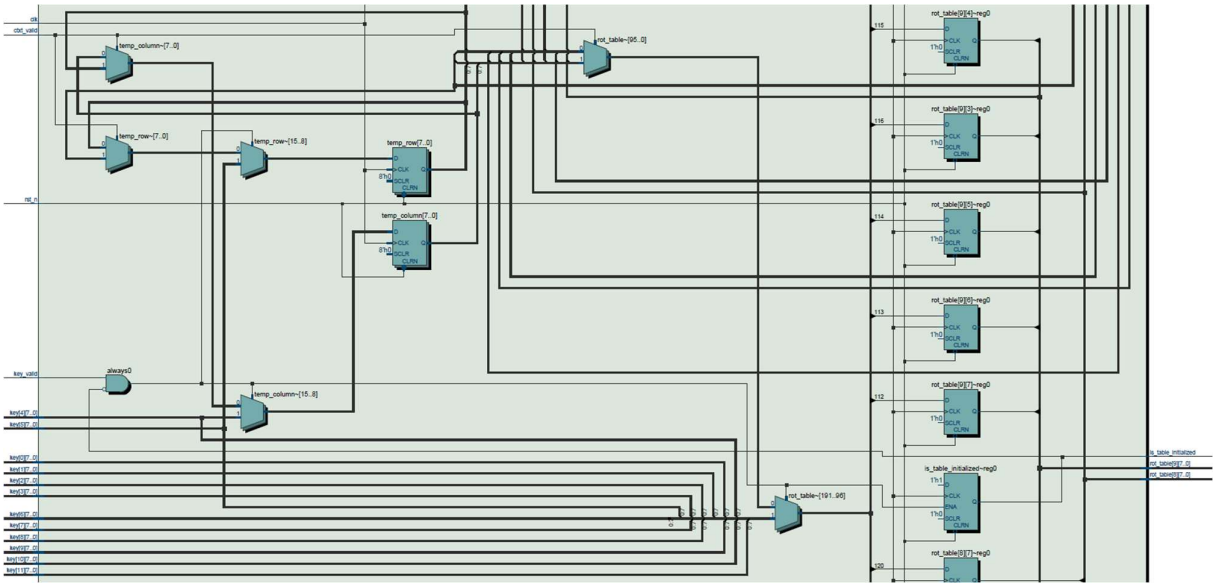

Figure 4. Architecture of the rot_table module.

Figure 4 shows a part of the module *rot_table*. In particular *temp_row* and *temp_column* are two registers used in the rotation phase to store the bytes that comes out of the array as a result of the shift operation.

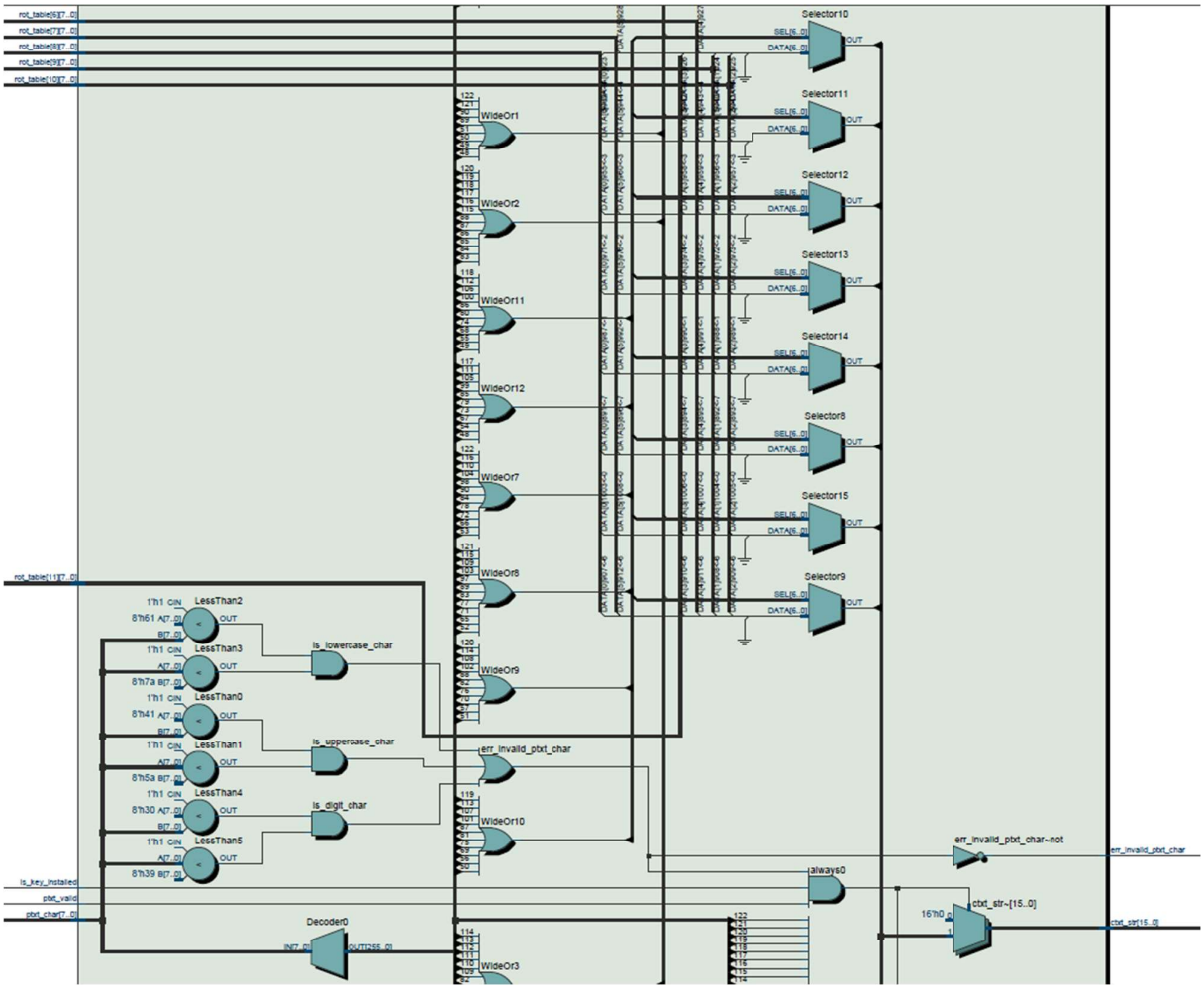

Figure 5. Architecture of the substitution_law module.

Figure 5 shows in the bottom area the computation of *ctxt_str*, that is driven by the flags *ptxt_valid*, *is_key_installed* and *err_invalid_ptxt_char*. Particularly, when those flags are respectively equals to 1, 1 and 0 then the ciphertext can be calculated.

3. Expected waveforms

This chapter shows the most relevant parts of the waveform.

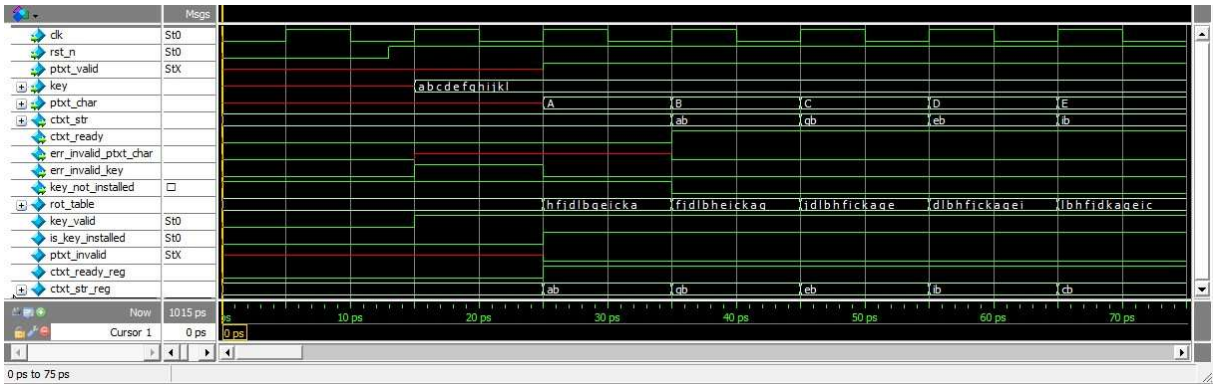

Figure 6. Initial part of the TEST_WORKFLOW waveform.

In Figure 6 is depicted a complete workflow of the *rst_cipher* module. After the *rst_n* signal switches from logic value 0 to 1, the RST is initialized with the key "*abcdefghijkl*" and this takes one clock cycle. Then the plaintext "A" is taken in input and at the next clock cycle the correspondent ciphertext is available to the *ctxt_str* as "ab", with signal *ctxt_ready* equals to 1. After encryption, the RST is rotated, indeed the next plaintext "b" is correctly encrypted with "gb".

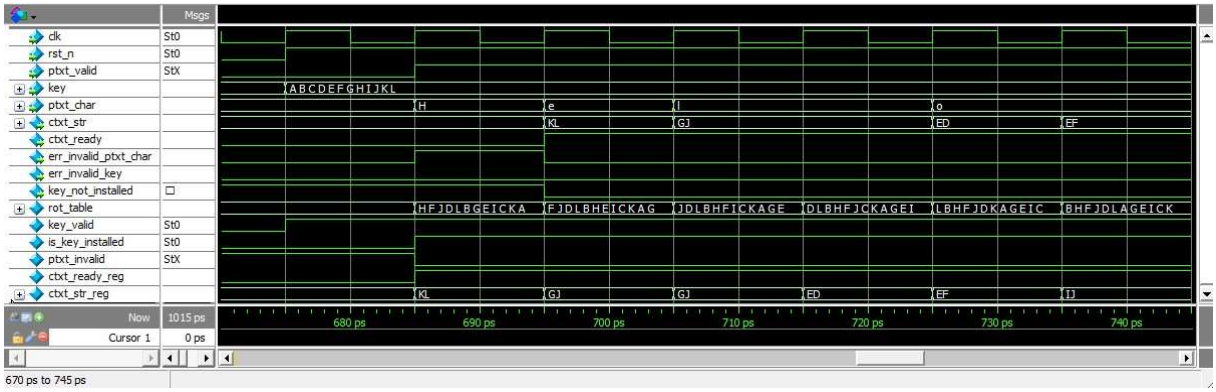

Figure 7. HELLO_EXAMPLE waveform.

Figure 7 shows the simulation of the example provided in the project specifications. After key initialization as "ABCDEFGHIJKL", the "Hello" plaintext is correctly encrypted with "KLGJGJEDEF".

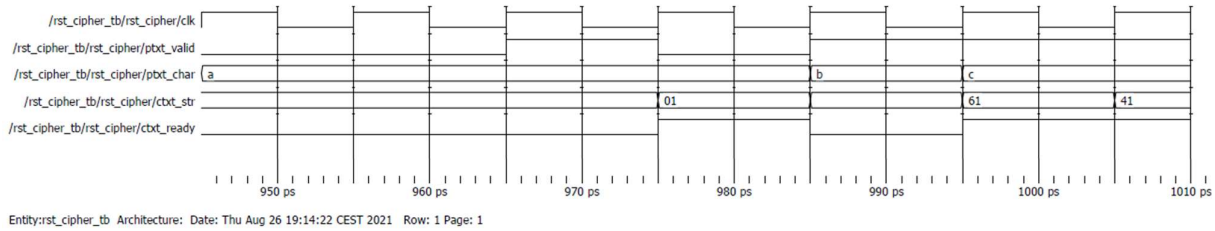

Figure 8. Waveform that shows the right behavior of signal ctxt_ready and ptxt_valid.

Figure 8 shows that when *ptxt_valid* is equal to 1 at 965 ps, the plaintext "a" is correctly processed by the *substitution_law* module, which generates "01" as ciphertext (key was "0123456789ab"), with the *ctxt_valid* signal equals to 1, representing that *ctxt_str* is consistent. At 975 ps, *ptxt_valid* goes to 0 and, as consequence, the plaintext "b" is not encrypted and *ctxt_ready* is 0, until 985 ps, when *ptxt_valid* returns to logic value 1.

This waveform is coherent with those ones indicated by the project specifications, depicted below by Figure 9:

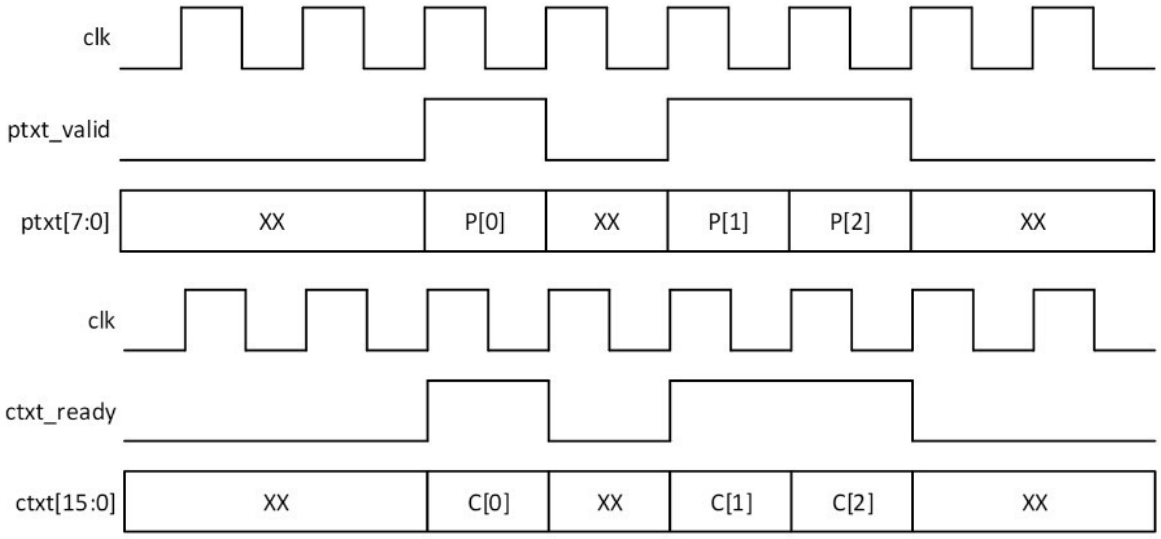

Figure 9. Target waveforms concerning ctxt_ready and ptxt_valid signals.

The following waveform shows the behaviour of the module in corner case scenarios.

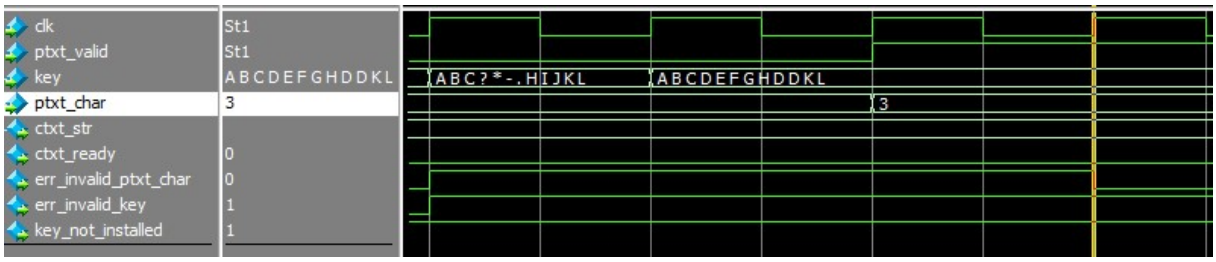

Figure 10. Waveform when key is not valid.

As depicted in figure 10, when the key is not valid the error flag *err_invalid_key* switches to 1, and of course also *key_not_installed* is 1. So, when the key is not valid and it is passed a plaintext (in this case "3") and *ptxt_valid* is 1, the *ctxt_ready* is correctly set to 0, indicating that the ciphertext is not valid.

The next corner case is shown in the waveform below (Figure 11). The clock cycle following the valid input key shows that flag *key_not_installed* goes to 0, but also the *ptxt_valid* do the same. In this case the plaintext is still "3", but no ciphertext is generated since the *ptxt_valid* flag is not 1.

Figure 11. Waveform when ptxt_valid is 0 and key is installed.

4. Testbench

This chapter describes the implemented testbench and its purpose.

TEST_WORKFLOW: This test aims to check the behaviour of the module when a very long message arrives at the input. This test verifies the correct behaviour of the module in rotations, substitutions and in the correct assignment of the regularity flags made at each clock cycle.

HELLO_EXAMPLE: This test aims to simulate the example provided in the project specification and confirm the expected result.

TEST_ERROR: The purpose of this test is to check the behaviour of the module in presence of corner cases and verify the expected assignments of error flags. In sequence the case of incorrect input key, non-installed key, invalid input plaintext, test of rotation of valid plaintext following an invalid one was tested.

WAVEFORMS: The purpose of this test is to simulate the waveforms required by the design specification for the *ptxt_valid* and *ctxt_ready* regularity flags.

5. Implementation of RTL design on FPGA and results

This section contains the steps of Analysis & Synthesis and Fitter that have been performed using Quartus. These reports show that the chosen FPGA is from the Cyclone V family. Devices in this family have low costs and low power consumption.

The Fitter Resource usage summary report displays a detailed analysis of logic utilization based on calculations of ALM usage. Logic utilization is the metric for the number of ALMs necessary to implement the design, in this case the RST cipher design, displayed as a fraction of the total ALMs available on the target device (ALMs needed / total ALMs on the device). The final report displays logic utilization lower than 1% meaning a low usage of the overall resources available on the target device.

Flow Status	Successful - Sat Aug 28 16:07:01 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	test_project
Top-level Entity Name	rst_cipher
Family	Cyclone V
Device	5CGXFC9D6F27C7
Timing Models	Final
Logic utilization (in ALMs)	390 / 113,560 (< 1 %)
Total registers	148
Total pins	1 / 378 (< 1 %)
Total virtual pins	126
Total block memory bits	0 / 12,492,800 (0 %)
Total DSP Blocks	0 / 342 (0 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 17 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 12. Summary of resource usage after Fitting.

The number of registers amount to 148 and there is a number of virtual pins (126) that corresponds to the to the effective code implementation (the real-one pin is the clock *clk*):

- *rst_n* 1-bit
- *ptxt_valid* 1-bit
- *ctxt_ready* 1-bit
- *err_invalid_ptxt_char* 1-bit
- *err_invalid_key* 1-bit
- *key_not_installed* 1-bit
- *ptxt_char* 8-bit
- *ctxt_str* 16-bit
- *key* 96-bit

6. Static Timing Analysis (STA)

For the timing analysis, a time constraint file (SDC1.sdc) was defined by specifying a target frequency of 100MHz, defining a clock with a period of 10 nanoseconds. In addition, for input/output signals, a maximum and minimum delay was defined, 20% and 10% of the system clock respectively.

In Figure 11 are the frequencies obtained during the slow tests for 85C and 0C respectively:

Fmax	Restricted Fmax	Clock Name	Note
135.46 MHz	135.46 MHz	clk	

Slow 1100mV 85C Fmax

Fmax	Restricted Fmax	Clock Name	Note
134.99 MHz	134.99 MHz	clk	

Slow 1100mV 0C Fmax

Figure 13. Comparison of Fmax during slow test at 85C and 0C.

```
create_clock -name clk -period 10 [get_ports clk]
set_false_path -from [get_ports rst_n] -to [get_clocks clk]
set_input_delay -min 1 -clock [get_clocks clk] [get_ports {rst_n ptxt_valid key[*][*] ptxt_char[*]}]
set_input_delay -max 2 -clock [get_clocks clk] [get_ports {rst_n ptxt_valid key[*][*] ptxt_char[*]}]
set_output_delay -min 1 -clock [get_clocks clk] [get_ports {ctxt_str[*] ctxt_ready err_invalid_ptxt_char err_invalid_key key_not_installed}]
set_output_delay -max 2 -clock [get_clocks clk] [get_ports {ctxt_str[*] ctxt_ready err_invalid_ptxt_char err_invalid_key key_not_installed}]
```

Figure 14. Content of .sdc file.

Figure 14 shows the *.sdc* file developed with several time constraints.

The *create_clock* command has the function of creating a clock object with a 10 ns period on the input port of the module that concerns the clock signal.

The *set_false_path* command specifies a false path for *rst_n* signal, in order to avoid time constraints checking with the previously created clock.

In the rest of the *.sdc* file there are four more commands, *set_input_delay* and *set_output_delay*, which represent constraints regarding input and output ports of the module.