UNIVERSITÀ DI PISA

Laurea magistrale in Cybersecurity (LM-66)

Esame di Hardware and Embedded Security

3-stage Caesar Cipher

Studenti:

Leonardo Talerico

Davide Morucci

Professori:

Sergio Saponara

Stefano Di Matteo

Specification analysis

The project requires the development of a hardware module capable of implementing an advanced version of the Caesar's cipher mechanism.

$$C[i] = CS_{K3,d3}(CS_{Kx,dx}(CS_{K1,d1}(P[i])))$$

This new mechanism consists in the application of three movements controlled by two directions (left or right) and two values (between 1 and 26) entered as input by the user and by a direction and a value deriving from the other two according to the following formulas:

$$Kx = (K1 + K3)\ mod\ 27\ ,\ dx = d1 \oplus d3$$

Where Kx is the value, dx is the direction, K1 and K3 are the values passed in input specifying the dimensions of the movements and d1 and d3 are the directions of the other movements.

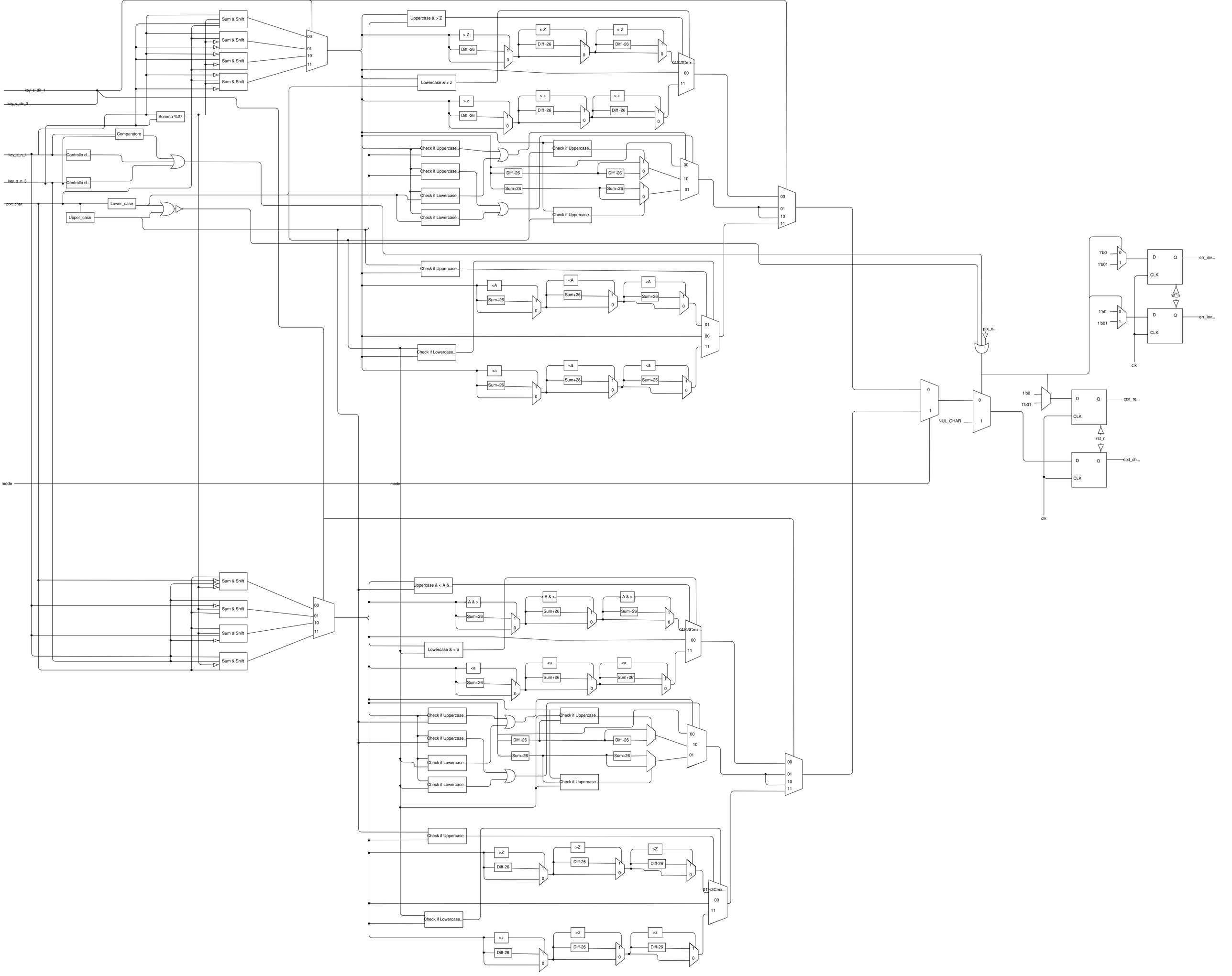
As such, the sequential application of the Caesar cipher mechanism uses a different direction and value in each of the three steps. It is necessary that the displacement values to be passed as input are different.

The mechanism deals with encrypting / decrypting (option chosen from a specific value passed in input) only characters between A-Z and a-z of the ASCII alphabet, returning instead the NULL character if the character supplied in input does not belong to these intervals.
In addition, the encryption / decryption of the character also depends on an additional input that informs the validity of the input character and therefore requires its processing.

Block diagram and design choices (RTL design module)

Description

The methodology chosen to address the problem was to divide it into two sections: the first section (the upper one in the diagram) used for encryption and the second (the lower one) to perform the decryption. There is also a multiplexer placed at the end of the two sections driven by the mode input to determine which of the two roads need to pass.
In addition, a splitting mechanism was used for both encryption and decryption. In fact, since there are only 4 possibilities of action for both cases, all the possible ciphers that can be carried out with the shift values provided in input are carried out and then a multiplexer driven by the two input wires d1 and d2 decides which value need to pass. These shifts are essentially comparable to sums or subtractions between the past values and the one obtained, so we have been able to simplify the single operations to the detriment of a higher hardware dimension [fig.1].



Viewer does not support full SVG 1.1

Input

The inputs provided to the system are as follows:

- clk
  - signal used to manage the registers used to provide the outputs
- rst_n
  - signal used to reset the registers used to provide the outputs
- ptxt_valid
  - signal used to highlight when the character supplied in input is ready to be encrypted/decrypted
    - 1'b0 = wait
    - 1'b1 = run
- mode
  - signal used to choose whether to perform encryption or decryption
    - 1'b0 = encrypt
    - 1'b1 = decrypt
- key_shift_dir_1
  - signal used to highlight the first shift direction
    - 1'b0 = right shift
    - 1'b1 = left shift
- key_shift_dir_3
  - signal used to highlight the third shift direction
    - 1'b0 = right shift
    - 1'b1 = left shift
- key_shift_num_1
  - 5-bit signal used to highlight how much to perform the first shift
- key_shift_num_3
  - 5-bit signal used to highlight how much to perform the third shift
- ptxt_char
  - 8-bit signal that represents the character on which to act

Output

The outputs provided by the system are the following:

- ctxt_char
  - 8-bit register containing the result of the encryption / decryption
- err_invalid_key_shift_num
  - one-bit register used to highlight the correctness or otherwise of the shift values passed in input. The values must be less than or equal to 26 and different from each other
    - 1'b0 = ok
    - 1'b1 = error
- err_invalid_ptxt_char
  - one-bit register used to highlight the fact that the character passed in input is not an uppercase or lowercase letter of the alphabet
    - 1'b0 = ok
    - 1'b1 = error

- ctx_ready
    - one-bit register used to highlight when the output value from ctxt_char is valid and can be used

Diagram's details

Encryption:

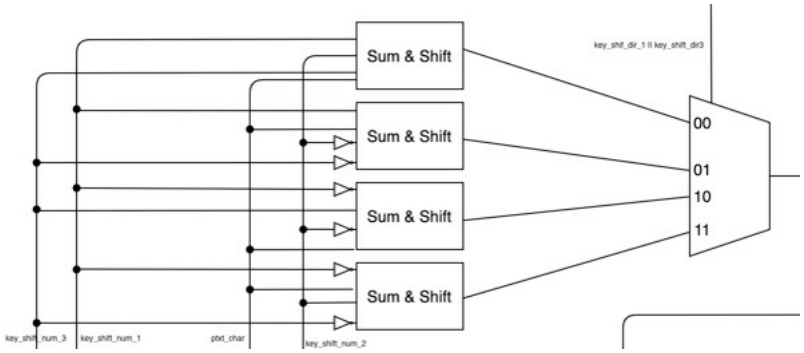


Fig. 2 Encryption block

The encryption is carried out in parallel by analyzing the only four possible encryption possibilities. In fact, they depend on the two directions entered by the user as input, as the second direction depends on the two previous ones. In this way it was possible to isolate the four cases representing the cases of addition / subtraction between the three shift values:

- d1 = 0, d3=0 -> dx = 0 -> ctxt = ptxt + shift1 +shift2 +shift3
- d1 = 0, d3=1 -> dx = 1 -> ctxt = ptxt + shift1 -shift2 -shift3
- d1 = 1, d3=0 -> dx = 1 -> ctxt = ptxt - shift1 -shift2 +shift3
- d1 = 1, d3=1 -> dx = 0 -> ctxt = ptxt - shift1 +shift2 -shift3

There is also a 4-way multiplexer controlled by wires d1 and d3 to pass one of the four operations performed [fig. 2].

Decryption:

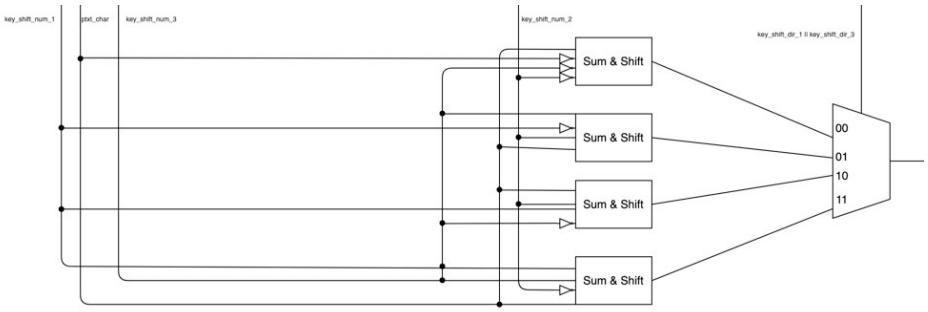


Fig. 3 Decryption block

Similarly, for the encryption there are four parallel hardware blocks carrying out the four possible decryptings, consider that in this case the keys are the same (i.e. the directions and the shifts) as the encryption, but the operations are opposite:

- d1 = 0, d3=0 -> dx = 0 -> ctxt = ptxt - shift1 -shift2 -shift3
- d1 = 0, d3=1 -> dx = 1 -> ctxt = ptxt - shift1 +shift2 +shift3
- d1 = 1, d3=0 -> dx = 1 -> ctxt = ptxt + shift1 +shift2 -shift3

- d1 = 1, d3=1 -> dx = 0 -> ctxt = ptxt + shift1 -shift2 +shift3

Also in this case there is also a four-way multiplexer [fig.3].

Control and modification circuitry in the event of a leak:

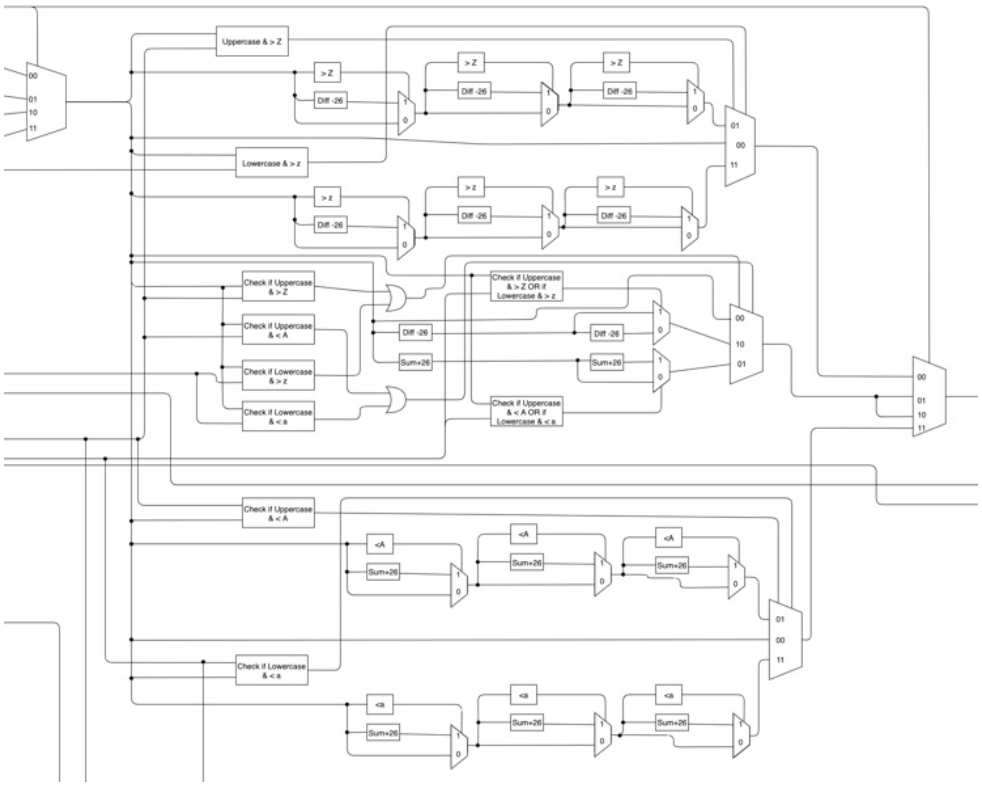


Fig. 4 Control block for encryption

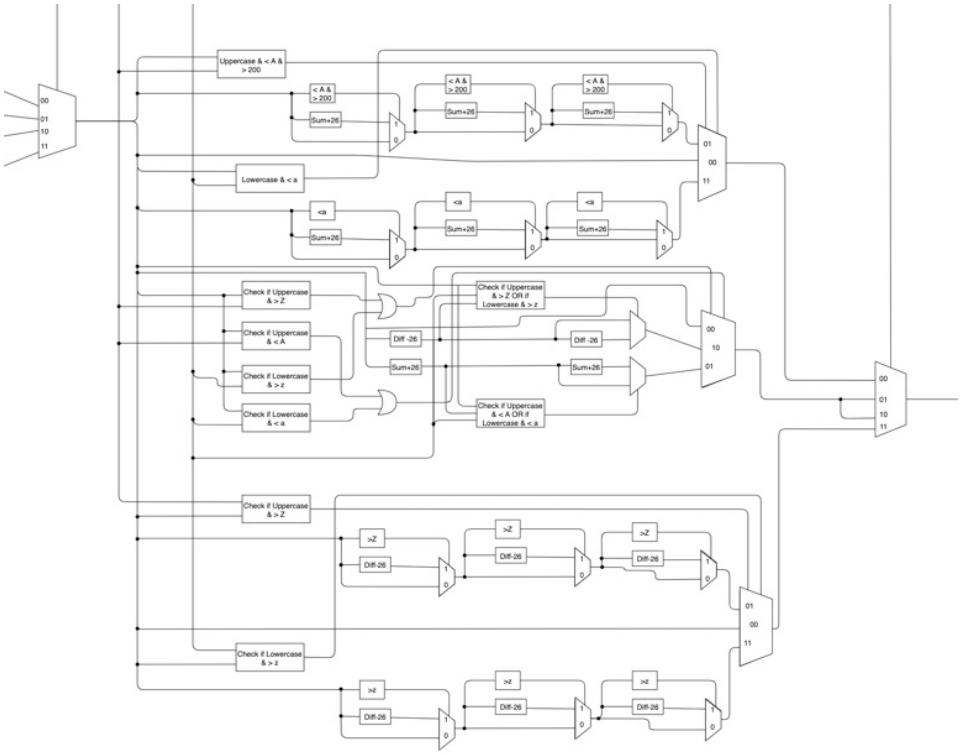


Fig. 5 Control block for decryption

The the necessary circuitry to carry out checks on the value reached is at the output of the multiplexers used to carry out the sums/subtractions to encrypt. This is necessary to ensure that in the event that values lower than a-A or higher than z-Z are obtained, the correct changes are applied in order to obtain the circularity of the encryption. In most cases it will be necessary to add/subtract 26 or 52 from the value obtained, while in the cases of encryption/decryption with d1 = 0 and d3 = 0 this value to be added/subtracted can also be 78 [fig.4].
In particular, in the case of decryption d1 = 0 and d3 = 0, the control is extended to verify the value greater than 200, because in this case the subtraction from ptxt could become less than 0 in the case of capital letters and therefore become a larger positive number (this problem is not present in encryption) [fig.5].

Choice of encryption/decryption and provision of value:

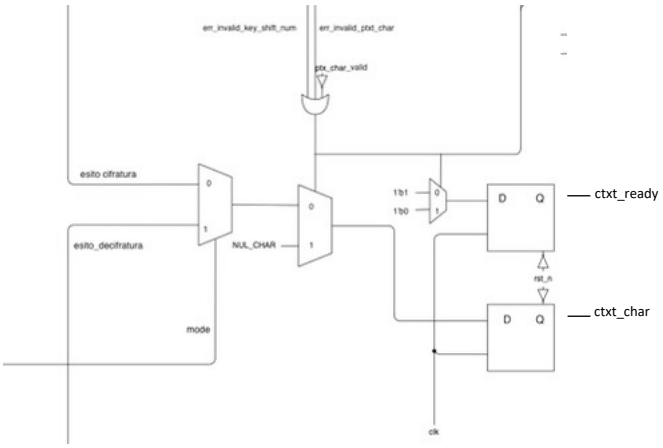


Fig. 6 Choice of encryption/decryption and output of value and correctness signal

Encryption and decryption are both carried out in parallel and at the end of their combinational chain there is a multiplexer managed by the wire mode:

- mode = 0 -> encryption
- mode = 1 -> decryption

Before the output register, there is also another multiplexer managed by the error signals to pass either the value just calculated or the null character [fig.6]

Generation of alarm signals:

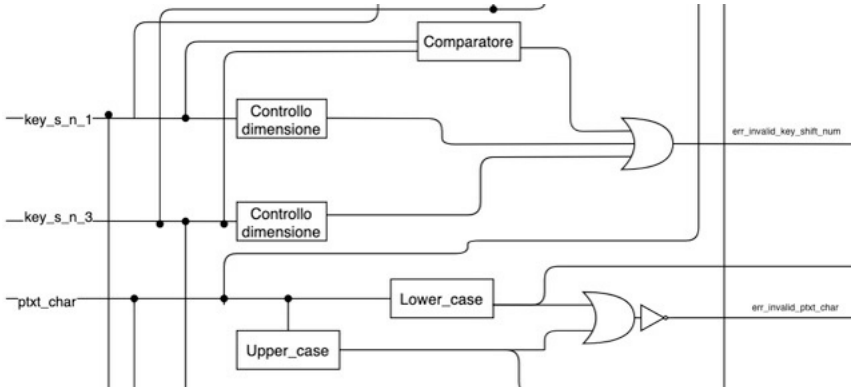


Fig. 7 Generation of error signals

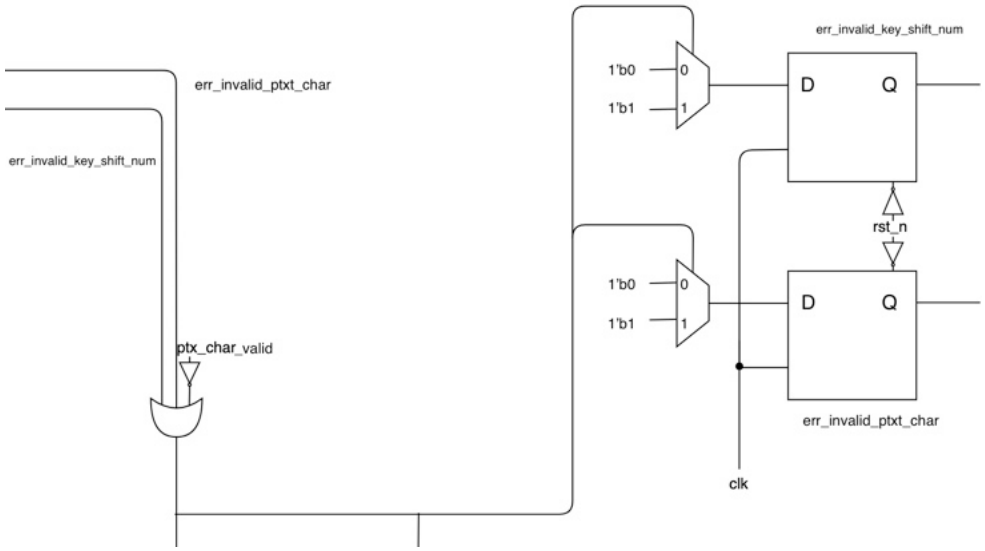


Fig. 8 Error signal output

There are two warning signs err_invalid_key_shift_num and err_invalid_ptxt_char.
The first is generated by the OR of three checks:

- shift1 > 26
- shift3 >26
- shift1==shift3

In the event that even only one of these conditions is verified, this signal is placed at one highlighting the error.
The second signal is generated by the negated OR of the ptxt checks regarding its belonging to lowercase or lowercase characters:

- ptxt is lowercase
- ptxt is uppercase

These controls return 1 if the character belongs to the intervals [65:90] or [97: 122] of the ASCII alphabet. The negated OR therefore returns 1 only if both are 0 and therefore the character is not a letter of the alphabet [fig.7].
These two signals and the negated signal of ptxt_ready are then inserted into an OR which manages 2 multiplexers and whose outputs are connected to registers for the outputs [fig.8].

Testbench (design choices, comments and C model)

C model

The cipher under analysis was developed using the C programming language. This was done in order to have a model that can be used as a comparison in verifying the correct functioning of the implemented hardware. The choice of this language is mainly due to the speed of implementation obtained over the years and to the accumulated experience that has allowed everything to be developed in a short time. Furthermore, the language also provides functions for reading and writing from files used for testing.
Program C requires the user to enter five parameters as input:

- mode -> 0 decryption and 1 encryption
- first direction -> 0 left and 1 right

- third direction -> 0 left and 1 right
- value of the first shift
- value of the third shift

Once the inputs have been entered and the program executed, in case of encryption, it will generate a random string and write it in the prova.txt file and write the relative encryption in the enc.txt file, while in the case of decryption it will read the enc.txt file and write the relative decryption in the file dec.txt.
The program implements exactly the same measures on dimensions and constraints imposed on the hardware device. This means that in case of errors it will not encrypt / decrypt the character, but will insert the NULL character as per specification.
The process is done twice with different file names, and with different dimension of the string that is generated.

Testbench in System Verilog

In this case there are two testbench modules:

- caesar_ciph_tb_checks
    - This module carries out various tests on all characters of the alphabet by printing on the screen each time the value of the corresponding encryption / decryption in order to observe the behavior of the device and ensure its correct functioning in different cases.
      The cases analyzed are:
        - shift_dir_1 = 1'b1; shift_dir_3 = 1'b0; shift_N_1 = 5'd25; shift_N_3 = 5'd5; input_valid = 1'b1; mode = 1'b0;
        - shift_dir_1 = 1'b0; shift_dir_3 = 1'b0; shift_N_1 = 5'd25; shift_N_3 = 5'd26; input_valid = 1'b1; mode = 1'b1;
        - shift_dir_1 = 1'b0; shift_dir_3 = 1'b1; shift_N_1 = 5'd5; shift_N_3 = 5'd6; input_valid = 1'b1; mode = 1'b0;
        - shift_dir_1 = 1'b1; shift_dir_3 = 1'b0; shift_N_1 = 5'd10; shift_N_3 = 5'd2; input_valid = 1'b1; mode = 1'b1;
        - shift_dir_1 = 1'b1; shift_dir_3 = 1'b1; shift_N_1 = 5'd5; shift_N_3 = 5'd20; input_valid = 1'b1; mode = 1'b0;
        - shift_dir_1 = 1'b0; shift_dir_3 = 1'b0; shift_N_1 = 5'd28; shift_N_3 = 5'd1; input_valid = 1'b1; mode = 1'b1;

      In particular, the last test allows you to observe the behavior of the device in the event that the condition on the size of the shift value is violated.
- caesar_ciph_tb_file_enc

  This module performs tests on the test.txt and test2.txt files located in the modelsim/model folder. The test, in fact, opens the first encrypted file character by character, writes it in the modelsim/tv/enc_HW.txt file and also inserts the encrypted character in a buffer as long as it is correct, otherwise NULL, therefore starting from the first character in the file modelsim/tv/enc_HW.txt starts decrypting it and writes it to modelsim/tv/dec_HW.txt file and another buffer. Once the operation is completed, the modelsim/model/enc.txt file containing the file encrypted by model C is opened and inserted in a buffer, after which the same operation is performed with the modelsim/tv/dec.txt file . at the end, comparisons are made between the buffers containing the ciphers and decrypts to ensure that the C model and the hardware model are identical, returning the result to the screen.

Similarly, the same operations are performed also with the second file with the only difference in the name of the files modelsim/model/enc2.txt, modelsim/model/dec2.txt, modelsim/tv/enc_HW.txt and modelsim/tv/dec_HW .txt. The keys are:

- shift_dir_1 = 1'b1; shift_dir_3 = 1'b0; shift_N_1 = 5'd16; shift_N_3 = 5'd1; input_valid = 1'b1;
- shift_dir_1 = 1'b0; shift_dir_3 = 1'b0; shift_N_1 = 5'd25; shift_N_3 = 5'd26; input_valid = 1'b1;

Waveforms

After implementing a TestBench for our module, we could observe the following waveforms.

Encryption waveform

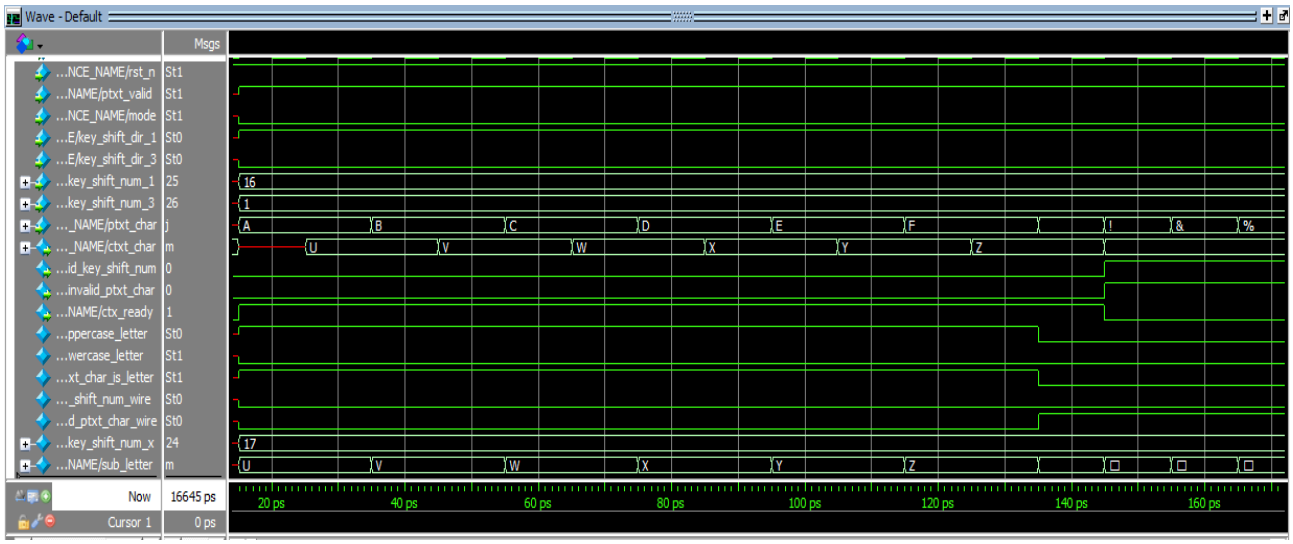


Fig. 9 Encryption waveform

The waveform shows a successful encryption example, using 16 and 1 as input key values, respectively for key_shift_num_1 and key_shift_num_3, with negative direction for the first (left) and positive for the second (right). As you can see the wire mode is set to 0, since we are in encryption. Below you can see the value 17, calculated by adding the values of key_shift_num_1 and key_shift_num_3, and marked with key_shift_num_x. In this case we can notice some aspects that distinguish the functioning of the circuit, in particular: the invalid_ptxt_char wire is set to 0 as we expected, until the first illegal value is reached, that is the empty space and the subsequent invalid characters (ex .!% &? ...), in this case the value transits to 1 to signal the invalidity and at the same time also ctx_ready transits to 0 causing the appearance of the NUL_CHAR value on the output register. Each single character encryption is performed in a clock cycle, as per specification [fig.9].

Decryption waveform

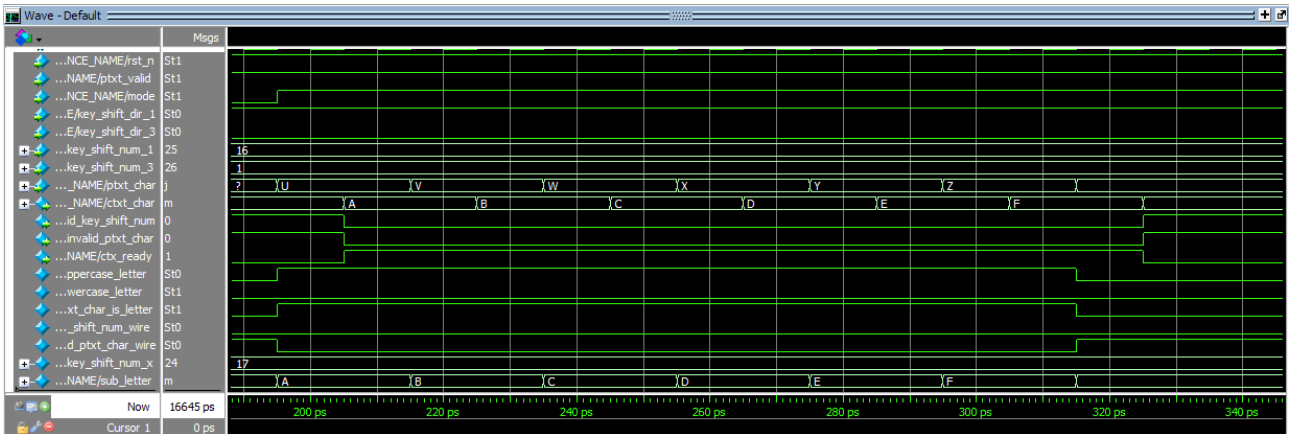


Fig. 10 Decryption waveform

A similar behavior to the previous case of Encryption can also be observed for the Decryption part. As we can see, also in this case I have a transition of the mode value, set to 1 to regulate the operating mode of the circuit. In this case we can focus on some of the values previously left out, in particular: as we expected, after the last valid character ("F" in this case), we have transitions regarding the values of invalid_ptxt_char and ptxt_char_is_letter, to signal the presence of an invalid value in input to the circuit. In such cases a NUL_CHAR value will be output again [fig.10].

Examples of waveforms in case of errors and checks

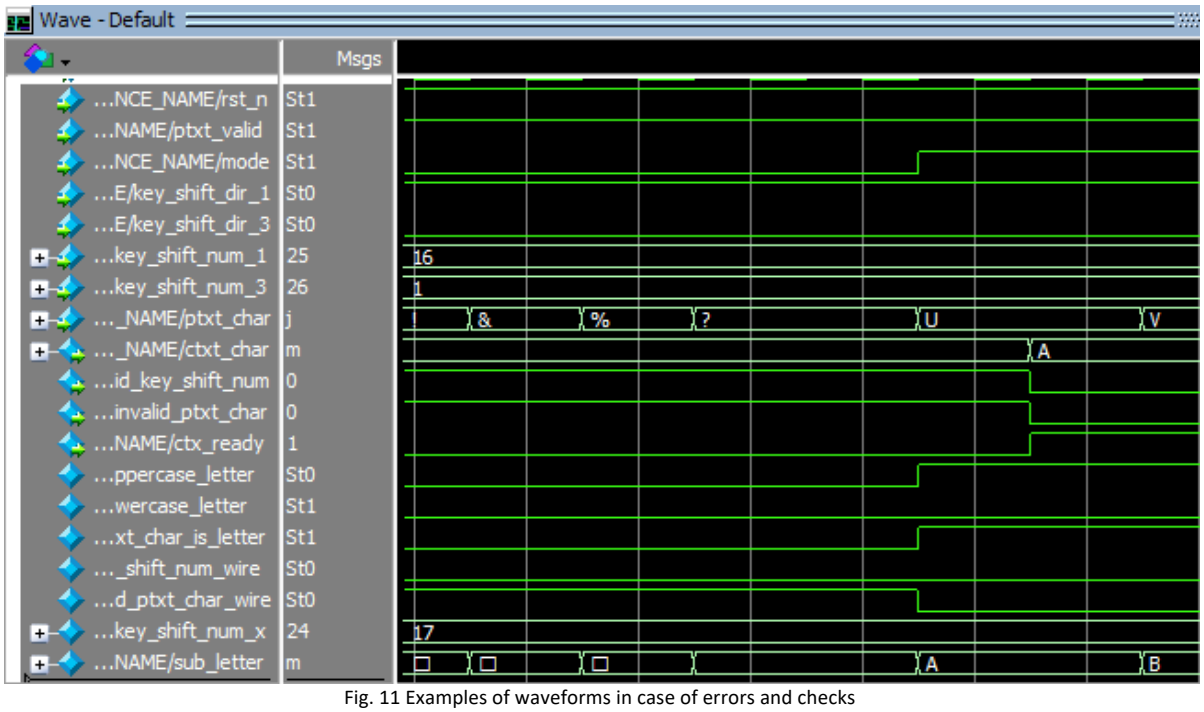


Fig. 11 Examples of waveforms in case of errors and checks

Finally, in this test we can look more carefully at the errors that may arise within our circuit, in the presence of unauthorized input values (as seen above). In this case we can also observe the value of invalid_key_shift_num which is set by default in the presence of invalid characters, while in a case of normal operation with valid characters, the value of the latter will pass to 1 if one of the two keys (key_shift_num_1 and key_shift_num_3) do not respect the constraint <= 26.

The other aspect reported in the waveform above concerns the Lowercase and Uppercase values, which are set to 0 in cases of invalid inputs or errors [fig. 11].

Implementation of RTL design on FPGA

Quartus summary

Flow Summary

<<Filter>>

Flow Status	Successful - Fri Sep 10 17:13:19 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	3_stage_caesar
Top-level Entity Name	caesar_cipher
Family	Cyclone V
Device	5CGXFC7D6F27C7
Timing Models	Final
Logic utilization (in ALMs)	424 / 56,480 (< 1 %)
Total registers	11
Total pins	1 / 378 (< 1 %)
Total virtual pins	34
Total block memory bits	0 / 7,024,640 (0 %)
Total DSP Blocks	0 / 156 (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 %)

Fig 12. Summary

The use of logic (in the ALMs) is less than 1% of the total FPGA resources [Fig. 12].
The 11 registers are divided as follows:

- 8 registers for ctxt_char
- 1 register for ctxt_ready
- 1 register for err_invalid_ptxt_char
- 1 registers for err_invalid_key_shift_num

The total number of pins is 1, because the only signal connected is the clock.
The 11 output registers mentioned above require 11 virtual pins. By adding the virtual input pins to them, the total number rises to 34, as the module requires:

- 8 input wires for ptxt_char
- 5 input wires for key_shift_num_1
- 5 input wires for key_shift_num_3
- 1 input wire for key_shift_dir_1
- 1 input wire for key_shift_dir_3
- 1 input wire per mode
- 1 input wire for ptxt_valid

- 1 input wire for rst_n

Netlist

Below is a zoom on the most relevant parts of the synthesis.

Check to see if ptxt is a character:

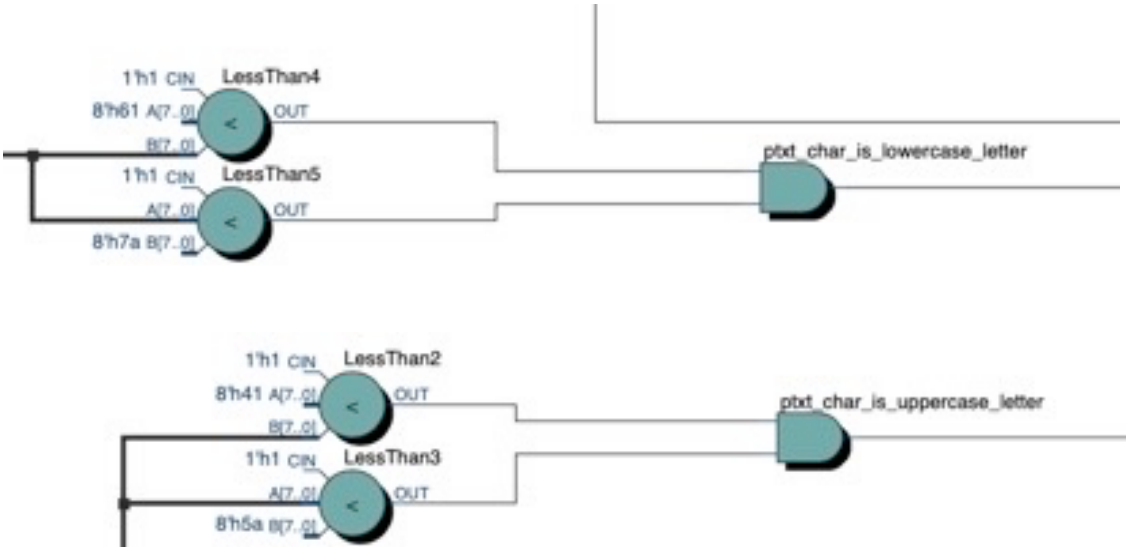


Fig 13. Check to see if ptxt is a lowercase character (above) and check to see if ptxt is an uppercase character (below)

In this case it is possible to observe how Quartus synthesized the part relating to the control of the inserted character. In fact, it is checked whether the value is included in the interval [A-Z] or if it is included in the interval [a-z]. The two output wires of the two AND gates are then inputs of an OR gate which will provide the ptxt_char_is_letter value used for the error signals [fig.13].

Creating the key_shift_num_x value:

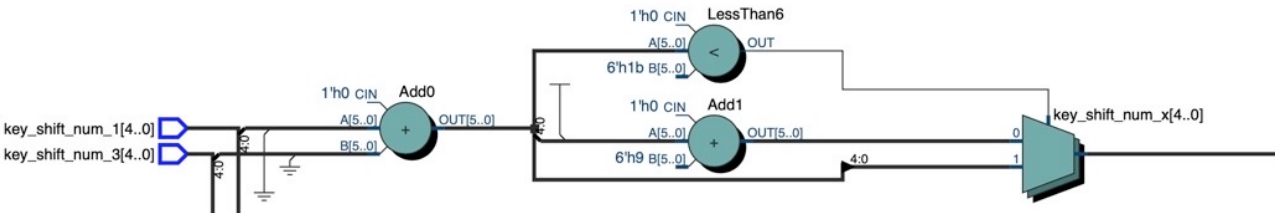


Fig 14. Creating the key_shift_num_x value:

In this case it is possible to observe how Quartus synthesized the part relating to the generation of the shift value of the second shift. The two shift values entered are in fact added together and subsequently, if the value obtained is greater than twenty-seven, this value is subtracted to perform the module [fig.14]

Generation of alarm signals:

Fig 15. Generation of the alarm signal that manages the error output registers

In this case it is possible to observe how Quartus has synthesized the part relating to the generation of the alarm signal which then manages the various error outputs. In fact, the ptxt_char_is_letter signal (negated signal at the bottom), the ptxt_valid input signal (also negated) and the results of the size and equality checks carried out on key_shift_num_1 and key_shift_num_3 [fig.15] enter the OR gate.

Passing the encryption/decryption value and output registers:

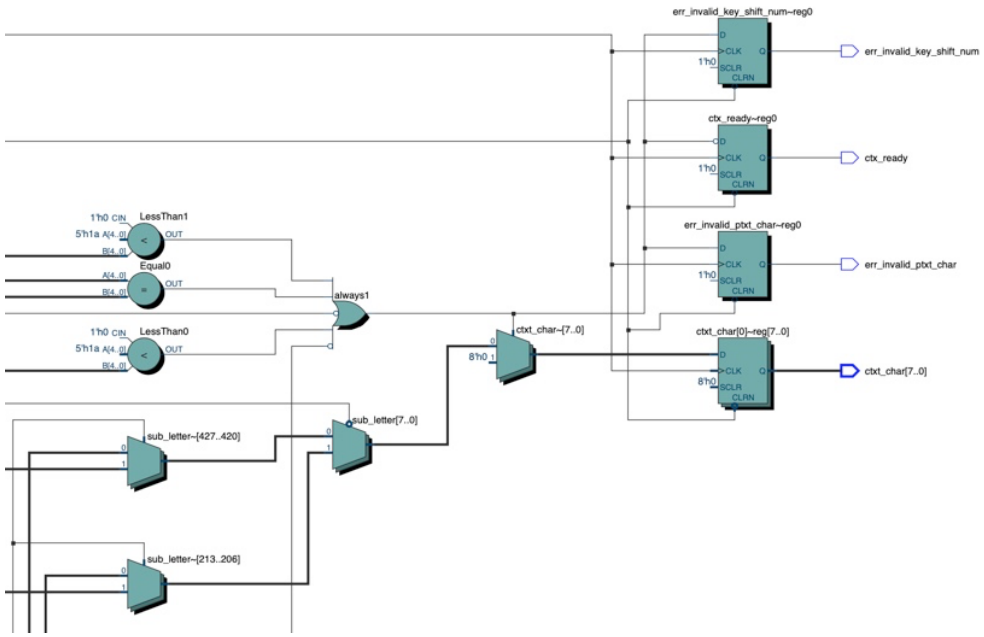


Fig 16. Passing the encryption / decryption value and output registers

In this case it is possible to observe how Quartus has synthesized the part relating to the outputs and the selection of the value to be given to ctxt. The first multiplexer on the left, in fact, on the basis of the mode signal, passes the result of the encryption or that of the decryption. The second multiplexer, on the other hand, takes care of passing the ctxt value or the NULL value in case of

some error. Finally, the first and third registers are managed by the wire resulting from the error signals, so as to release the value 1 or 0 according to the presence or absence of problems [fig. 16].

Static Timing Analysis

An SDC file was created, in which all the time constraints that the Quartus engine had to comply with were written [fig.17].

```
create_clock -name clk -period 5 [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 mode key_shift_dir_1 key_shift_dir_3 key_shift_num_1[*] key_shift_num_3[*] ptxt_char[*]}]
set_input_delay -max 2 -clock [get_clocks clk] [get_ports {rst_n ptxt_valid mode key_shift_dir_1 key_shift_dir_3 key_shift_num_1[*] key_shift_num_3[*] ptxt_char[*]}]
set_output_delay -min 1 -clock [get_clocks clk] [get_ports {ctxt_char[*] err_invalid_key_shift_num err_invalid_ptxt_char ctx_ready}]
set_output_delay -max 2 -clock [get_clocks clk] [get_ports {ctxt_char[*] err_invalid_key_shift_num err_invalid_ptxt_char ctx_ready}]
```

Fig 17. File SDC

These constraints include:

- Clock period
- Existence of an asynchronous reset
- Minimum / maximum entry and exit delays

To achieve at least a clock frequency of 100 MHz it was necessary to map the inputs and outputs not to the physical pins of the FPGA, but to the internal logic, assigning them to the virtual pins instead.
Therefore, with shorter I/O cables, the path delay was reduced and the frequency increased.
As regards the maximum frequency that the module can reach at the corner case 85C and 0C on the Slow 1100mV model, values of 64.41MHz and 61.9MHz respectively have been obtained [fig.18].

Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	64.41 MHz	64.41 MHz	clk	

Slow 1100mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name
1	61.9 MHz	61.9 MHz	clk

Fig. 18 F frequency reached at 85 C (left) and 0C (right)

These frequencies are particularly limited due to the considerable combinatorial logic present inside the device, which inevitably creates a propagation delay and forces the clock to be reduced to avoid malfunctions. Furthermore, this value could also be attributable to the device on which everything was synthesized, a value which on new devices could certainly be higher.
A solution to fix this frequency could be to introduce registers within the logic in order to create shorter paths and reduce any propagation delays. However, to comply with the design constraints provided, this was not possible because the encryption / decryption would no longer be performed during every clock cycle, but the supply would depend on the number of registers introduced.