UNIVERSITÀ DI PISA

Department of Information Engineering
Master's Degree in Cybersecurity

HARDWARE AND EMBEDDED SECURITY

REPORT
3-stage Caesar Cipher

Professors:
Sergio SAPONARA
Stefano DI MATTEO

Students:
Marco FRESCO
Emanuele URSELLI

Academic Year 2021/2022

Contents

1 Specification Analysis **3**

1.1 Expected Behaviour 4

1.2 Example 4

2 Block Diagram and Design Choices **6**

2.1 Block Diagram 6

2.2 RTL 7

3 Expected Waveforms **9**

4 Testbench **10**

4.1 Testbench Checks 10

4.2 File Encryption 11

5 Implementation of RTL design on FPGA and results **13**

6 Static Timing Analysis (STA) **16**

Chapter 1

Specification Analysis

The goal of the 3-Stage Caesar Cipher Project consists in the design and implementation of a derivation of one of the simplest and most widely known encryption techniques. The Caesar Cipher algorithm is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. For example, with a left shift of 3, 'D' would be replaced by 'A', 'E' would become 'B' and so on (fig. 1.1).

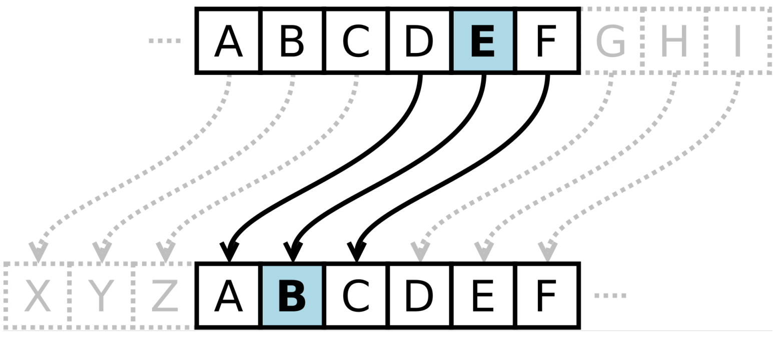


Figure 1.1: Caesar Cipher

The specifics of the project extends the requirements of the Caesar Cipher, enabling it to a 3-stage version, where the algorithm applies the same operations three times before giving as output the cipher character. The 3-stage Caesar Cipher can be summed up with the following formula:

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

- **CS** is the Caesar Cipher substitution algorithm
- **C[i]** is the 8-bit ASCII code of the ith character of ciphertext
- **P[i]** is the 8-bit ASCII code of the ith character of plaintext
- **Ki** is the ith Key (number of shift positions) to be used for the ith stage
- **di** is the ith shift direction (0 = right, 1 = left) for the ith stage

1.1 Expected Behaviour

To have a better understanding of the expected behavior of the designed hardware module, fig.1.2 and fig.1.3 can be taken into account. At every rising edge of the clock cycle, the module will encrypt/decrypt the passed character, with respect to the passed keys and the passed shift directions. Two more signals are crucial for the sampling process, which respectively are 'ctxt_valid' and 'ptxt_ready'.

- **ctxt_valid** is an input port which asserts the validity of the plaintext character passed as input.
- **ctxt_valid** is an output port which asserts the readiness of the ciphertext character to be consumed.

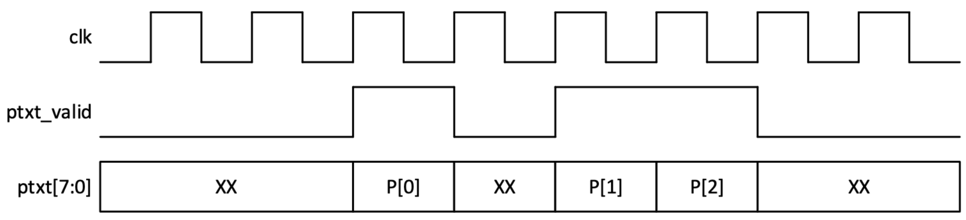


Figure 1.2: Analysis of Plaintext Signals

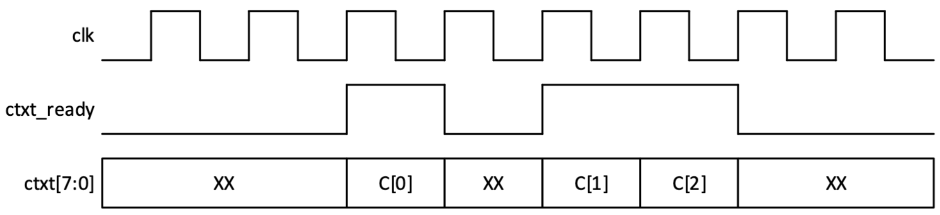


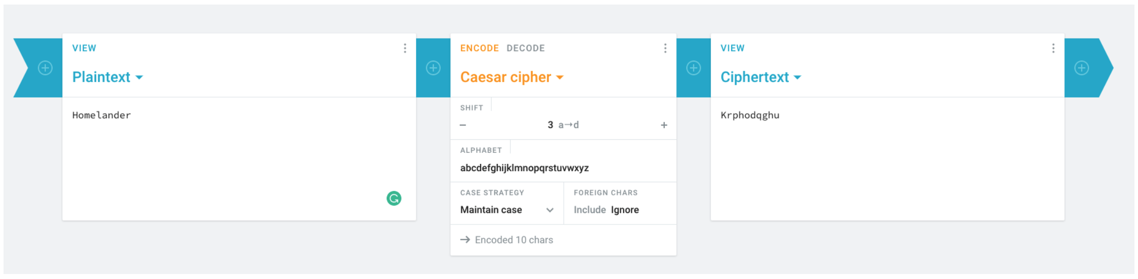
Figure 1.3: Analysis of Ciphertext Signals

1.2 Example

It's possible to imagine an applicable scenario concerning the 3-Stage Caesar Cipher of the subsequent form. First we set the keys and the shift direction,

$$< K1 = 3; K2 = 4; K3 = 5; D1 = Right; D2 = Right; D3 = Right >$$

Then we pass the plaintext characters from the word ”Homelander”. The three figures (fig.1.4, fig.1.5, fig.1.6) show the evolution of the algorithm along each stage of its application. Remember that case of each letter is maintained, and foreign characters are ignored.



Homelander

SHIFT 3 a→d

ALPHABET abcdefghijklmnopqrstuvwxyz

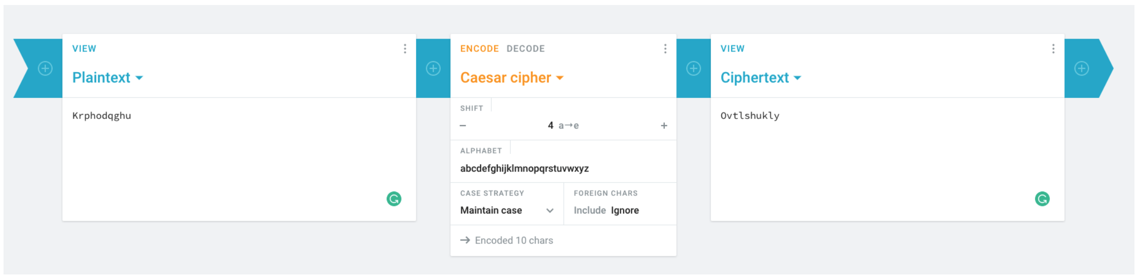
CASE STRATEGY Maintain case

FOREIGN CHARS Include Ignore

Encoded 10 chars

Krphodqghu

Figure 1.4: 1st Stage of Caesar Cipher (K1=3, D1=Right)



Krphodqghu

SHIFT 4 a→e

ALPHABET abcdefghijklmnopqrstuvwxyz

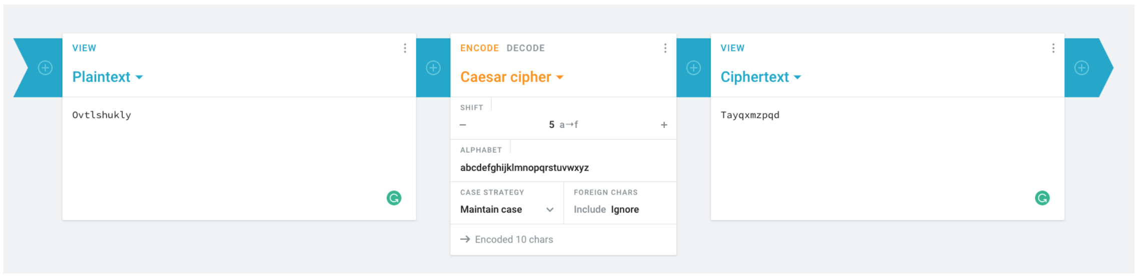
CASE STRATEGY Maintain case

FOREIGN CHARS Include Ignore

Encoded 10 chars

Ovtlshukly

Figure 1.5: 2nd Stage of Caesar Cipher (K2=4, D2=Right)



Ovtlshukly

SHIFT 5 a→f

ALPHABET abcdefghijklmnopqrstuvwxyz

CASE STRATEGY Maintain case

FOREIGN CHARS Include Ignore

Encoded 10 chars

Tayqxmzpqd

Figure 1.6: 3rd Stage of Caesar Cipher (K3=5, D3=Right)

Chapter 2

Block Diagram and Design Choices

2.1 Block Diagram

To have a high-level description of the hardware module designed, a suitable block diagram (fig.2.1) is provided as a facility for a sufficient understanding of the I/O Behavior.

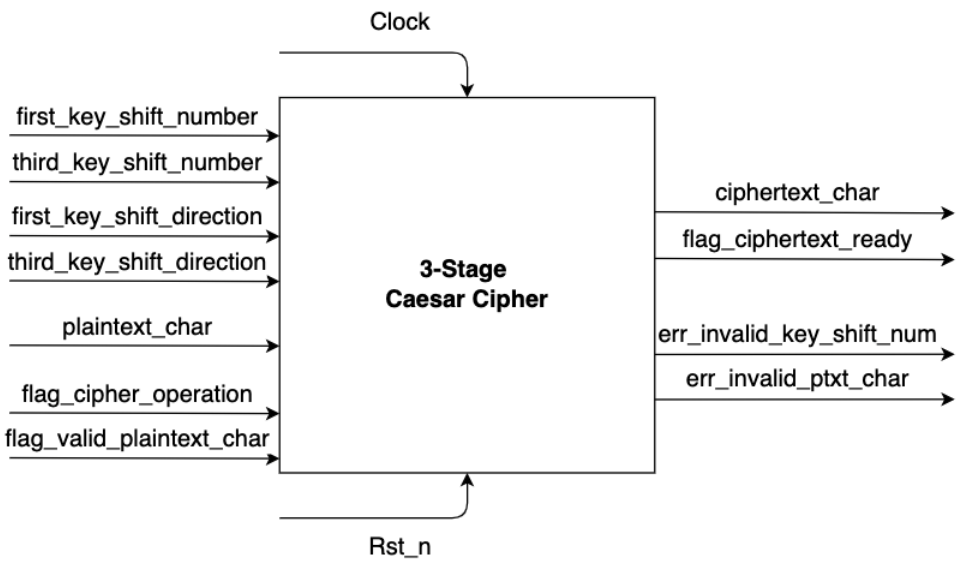


Figure 2.1: Block Diagram (High-Level View)

Concerning the input values, their semantics are the following:

- **first_key_shift_number**: Represents the first key K1, whose value indicates the number of positions to shift for the first stage of Caesar Cipher.

- **third_key_shift_number**: Represents the third key K3, whose value indicates the number of positions to shift for the third stage of Caesar Cipher.
- **first_key_shift_direction**: Represents the first direction D1, whose value specifies the shifting direction for the first stage of Caesar Cipher.
- **third_key_shift_direction**: Represents the third direction D3, whose value specifies the shifting direction for the thrid stage of Caesar Cipher.
- **plaintext_char**: Represents the character passed as input (usually a plaintext character, but may also be a ciphertext character, depending on the cipher mode)
- **flag_cipher_operation**: Represents the cipher mode (0 for encryption mode and 1 for decryption mode)
- **flag_valid_plaintext_char**: Represents the validity of the plaintext character passed as input to the hardware module (compliant to the ASCII standard values)

Concerning the output values, their semantics are the following:

- **ciphertext_char**: Represents the encrypted ciphertext character (or eventually the ciphertext decrypted), as the result of the processing performed by the 3-Stage Caesar Cipher module.
- **flag_ciphertext_ready**: Represents the validity of the encrypted ciphertext character (or the decrypted plaintext character), ready to be consumed for the next clock cycle.
- **err_invalid_key_shift_num**: Represents an error situation concerning a key quantity (invalid value of position to shift)
- **err_invalid_ptxt_char**: Represents an error situation concerning the plaintext character passed as input (invalid ASCII value)

2.2 RTL

After having given a black-box insight of the hardware module submitted, more considerations will be advanced for the underlying structure. A first phase of continuous assignments is performed, to define wires that either restrict the computation done to values that are within the domain of the Caesar Cipher specifications, or to values that are needed to perform the Caesar Cipher algorithm.

- **ptxt_char_is_uppercase_letter** and **ptxt_char_is_lowercase_letter** define the range of ASCII values that identify respectively uppercase and lowercase letters. **ptxt_char_is_letter** restricts the domain of each ASCII value to either an uppercase or lowercase letter.

- **flag_err_invalid_key_shift_num** and **flag_err_invalid_ptxt_char** both contribute in invoking error scenarios where the key quantities do not respect the specifics constraints or where the plaintext character has a value whose ASCII code does not match any letter (uppercase or lowercase).
- **second_key_shift_number** and **second_key_shift_direction** are respectively the second key K2 and the second shift direction d2, obtained as defined in the specifics as

  $$K2 = (K1 + K2) \bmod 27$$

  and

  $$d2 = d1 \oplus d3.$$

A second phase represented by the 'always' block deals with the effective calculus. It's noted that the encryption and decryption operations are symmetric. In particular, if for the 'encryption' case we perform sum operations when the shift direction encodes the 'right' direction, for the 'decryption' case we perform subtractions for the same shift direction, as we perform the reverse operation of those performed previously for encryption. Both in the case of sum and subtractions, some checks and eventual wrap operations are guaranteed to avoid 'overflow' or 'underflow' scenarios.

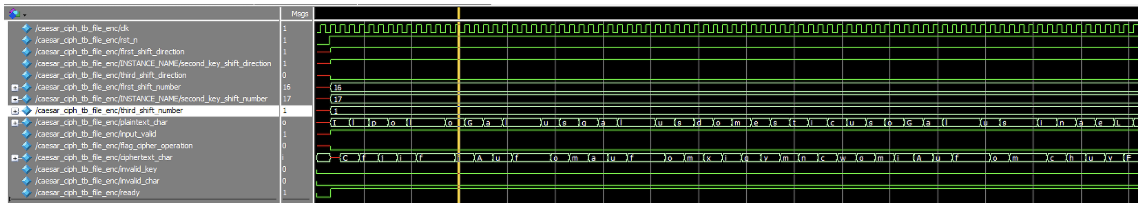
- **sub_letter** carries on the value of computations done along to the submitted plaintext, for each stage of the caesar cipher.

The third and final phase, represented by another 'always' block whose sensitivity list includes the positive edge of the clock and the negative edge of the asynchronous active-low reset, determines overall the exit status of the computation and the values calculated up until this moment.

Chapter 3

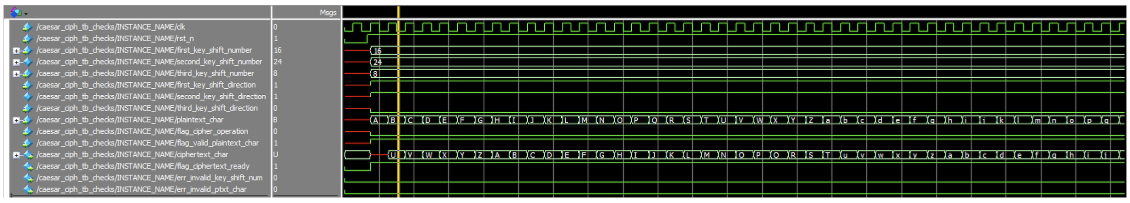
Expected Waveforms

In this section are shown two waveforms that represent the file encryption and the testbench checks. Before analyzing the waveforms, all signals in the diagrams are shown below. In the waveform of the file encryption we find the following signals:



```
/caesar_ciph_tb_file_enc/clk                                   1
/caesar_ciph_tb_file_enc/rst_n                                 1
/caesar_ciph_tb_file_enc/first_shift_direction                 1
/caesar_ciph_tb_file_enc/INSTANCE_NAME/second_key_shift_direction 1
/caesar_ciph_tb_file_enc/third_shift_direction                 0
/caesar_ciph_tb_file_enc/first_shift_number                    16
/caesar_ciph_tb_file_enc/INSTANCE_NAME/second_key_shift_number 17
/caesar_ciph_tb_file_enc/third_shift_number                    1
/caesar_ciph_tb_file_enc/plaintext_char                        o
/caesar_ciph_tb_file_enc/input_valid                           1
/caesar_ciph_tb_file_enc/flag_cipher_operation                 0
/caesar_ciph_tb_file_enc/ciphertext_char                       i
/caesar_ciph_tb_file_enc/invalid_key                           0
/caesar_ciph_tb_file_enc/invalid_char                          0
/caesar_ciph_tb_file_enc/ready                                 1
```

Figure 3.1: Waveform File Encryption



```
/caesar_ciph_tb_checks/INSTANCE_NAME/clk                         0
/caesar_ciph_tb_checks/INSTANCE_NAME/rst_n                       1
/caesar_ciph_tb_checks/INSTANCE_NAME/first_key_shift_number      16
/caesar_ciph_tb_checks/INSTANCE_NAME/second_key_shift_number     24
/caesar_ciph_tb_checks/INSTANCE_NAME/third_key_shift_number      8
/caesar_ciph_tb_checks/INSTANCE_NAME/first_key_shift_direction   1
/caesar_ciph_tb_checks/INSTANCE_NAME/second_key_shift_direction  1
/caesar_ciph_tb_checks/INSTANCE_NAME/third_key_shift_direction   0
/caesar_ciph_tb_checks/INSTANCE_NAME/plaintext_char              B
/caesar_ciph_tb_checks/INSTANCE_NAME/flag_cipher_operation       0
/caesar_ciph_tb_checks/INSTANCE_NAME/flag_valid_plaintext_char   1
/caesar_ciph_tb_checks/INSTANCE_NAME/ciphertext_char             U
/caesar_ciph_tb_checks/INSTANCE_NAME/flag_ciphertext_ready       1
/caesar_ciph_tb_checks/INSTANCE_NAME/err_invalid_key_shift_num   0
/caesar_ciph_tb_checks/INSTANCE_NAME/err_invalid_ptxt_char       0
```

Figure 3.2: Waveform Testbench Checks

An observation is pointed out to the values of the waves overall maintained. Starting from the second positive edge of the clock signal, for each clock cycle, the input character is sampled to produce the respective output character at the next positive edge of the clock. Since no anomalies have been registered concerning either the plaintext/ciphertext values or the key values, the error signals are 0 (false) and the validity flag of plaintext and ciphertext are as a consequence 1 (true). It's straightforward to deduce this from the two waves labelled respectively by the input and output character, where valid characters are presented for the whole duration of the wave.

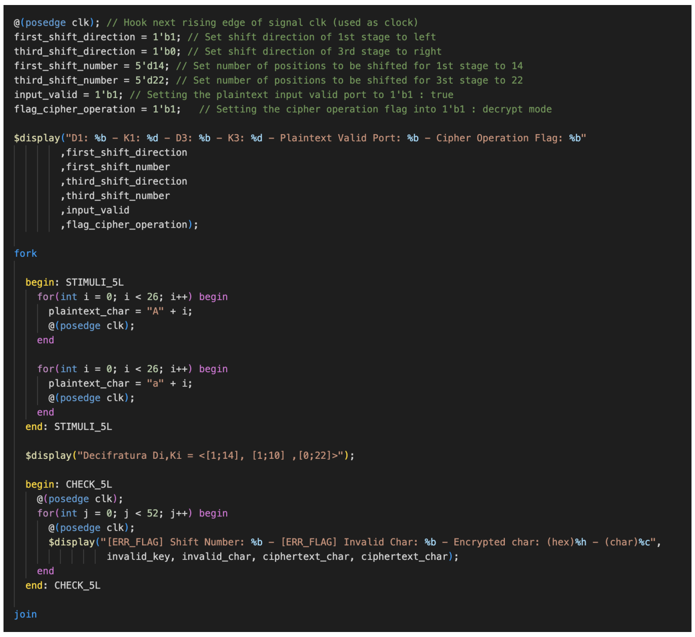
Chapter 4

Testbench

To test the designed module, two categories of test were written in the testbench file and simulated with the Modelsim Simulation Tool. The purpose of our tests was to verify that the module met the required specifications. In particular, the module shall encrypt (or decrypt) one plaintext (or ciphertext) character per clock cycle.

4.1 Testbench Checks

For the first round of tests, we submitted the alphabet range of letters (both uppercase and lowercase) to the processing done by the hardware module, in a very simple fashion.



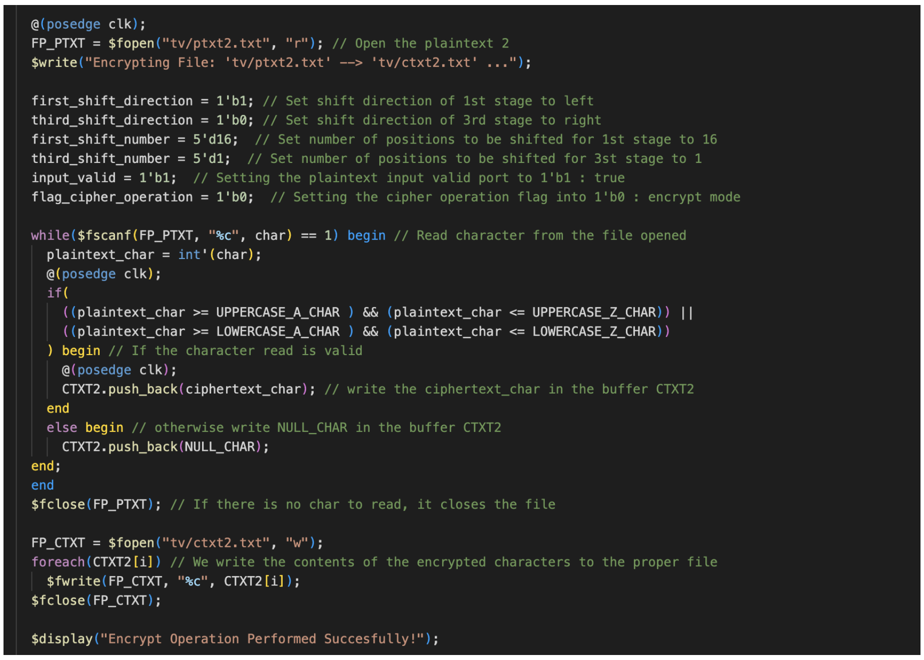
```
@(posedge clk); // Hook next rising edge of signal clk (used as clock)
first_shift_direction = 1'b1; // Set shift direction of 1st stage to left
third_shift_direction = 1'b0; // Set shift direction of 3rd stage to right
first_shift_number = 5'd14; // Set number of positions to be shifted for 1st stage to 14
third_shift_number = 5'd22; // Set number of positions to be shifted for 3st stage to 22
input_valid = 1'b1; // Setting the plaintext input valid port to 1'b1 : true
flag_cipher_operation = 1'b1;   // Setting the cipher operation flag into 1'b1 : decrypt mode

$display("D1: %b - K1: %d - D3: %b - K3: %d - Plaintext Valid Port: %b - Cipher Operation Flag: %b"
        ,first_shift_direction
        ,first_shift_number
        ,third_shift_direction
        ,third_shift_number
        ,input_valid
        ,flag_cipher_operation);

fork

  begin: STIMULI_5L
    for(int i = 0; i < 26; i++) begin
      plaintext_char = "A" + i;
      @(posedge clk);
    end

    for(int i = 0; i < 26; i++) begin
      plaintext_char = "a" + i;
      @(posedge clk);
    end
  end: STIMULI_5L

  $display("Decifratura Di,Ki = <[1;14], [1;10] ,[0;22]>");

  begin: CHECK_5L
    @(posedge clk);
    for(int j = 0; j < 52; j++) begin
      @(posedge clk);
      $display("[ERR_FLAG] Shift Number: %b - [ERR_FLAG] Invalid Char: %b - Encrypted char: (hex)%h - (char)%c",
              invalid_key, invalid_char, ciphertext_char, ciphertext_char);
    end
  end: CHECK_5L

join
```

Figure 4.1: Tasks of Testbench Checks

As it appears in fig.4.1, tasks have a similar structure where we store to a proper buffer structure all the letters of the alphabet, ready to be encrypted character by character. Before the task is performed, all the input parameters of the 3-stage Caesar Cipher module are setted in various ways, to exploit both the normal behavior (no anomalies concerning the I/O structure) and the abnormal ones (e.g. invalid key or char passed). The aimed goal was to showcase a very simple usage of the hardware module developed, and to have a fine-grained exploitable behavior to deploy it with more in-detail testing, such as in the case of the next category of test concerning file encryption and decryption.

4.2 File Encryption

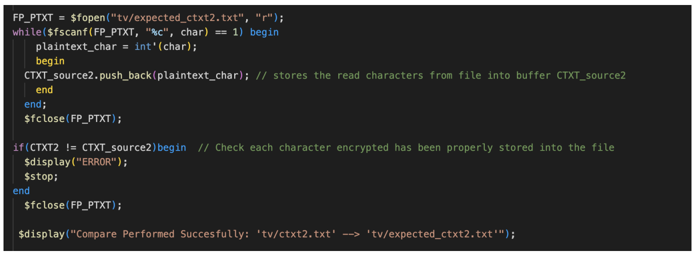
In the second round of tests, we adopted a more complex and realistic scenario dealing with the I/O behavior of files. We started from two samples of plaintext (respectively ptxt1.txt and ptxt2.txt files) and for both of them we submitted each of the characters in the files to our hardware module (fig.4.2), storing the corresponding ciphertext characters to ciphertext files (respectively ctxt1.txt and ctxt2.txt files).



```
@(posedge clk);
FP_PTXT = $fopen("tv/ptxt2.txt", "r"); // Open the plaintext 2
$write("Encrypting File: 'tv/ptxt2.txt' --> 'tv/ctxt2.txt' ...");

first_shift_direction = 1'b1; // Set shift direction of 1st stage to left
third_shift_direction = 1'b0; // Set shift direction of 3rd stage to right
first_shift_number = 5'd16;  // Set number of positions to be shifted for 1st stage to 16
third_shift_number = 5'd1;  // Set number of positions to be shifted for 3st stage to 1
input_valid = 1'b1;  // Setting the plaintext input valid port to 1'b1 : true
flag_cipher_operation = 1'b0;  // Setting the cipher operation flag into 1'b0 : encrypt mode

while($fscanf(FP_PTXT, "%c", char) == 1) begin // Read character from the file opened
  plaintext_char = int'(char);
  @(posedge clk);
  if(
    ((plaintext_char >= UPPERCASE_A_CHAR ) && (plaintext_char <= UPPERCASE_Z_CHAR)) ||
    ((plaintext_char >= LOWERCASE_A_CHAR ) && (plaintext_char <= LOWERCASE_Z_CHAR))
  ) begin // If the character read is valid
    @(posedge clk);
    CTXT2.push_back(ciphertext_char); // write the ciphertext_char in the buffer CTXT2
  end
  else begin // otherwise write NULL_CHAR in the buffer CTXT2
    CTXT2.push_back(NULL_CHAR);
end;
end
$fclose(FP_PTXT); // If there is no char to read, it closes the file

FP_CTXT = $fopen("tv/ctxt2.txt", "w");
foreach(CTXT2[i]) // We write the contents of the encrypted characters to the proper file
  $fwrite(FP_CTXT, "%c", CTXT2[i]);
$fclose(FP_CTXT);

$display("Encrypt Operation Performed Succesfully!");
```

Figure 4.2: File Encryption Operation

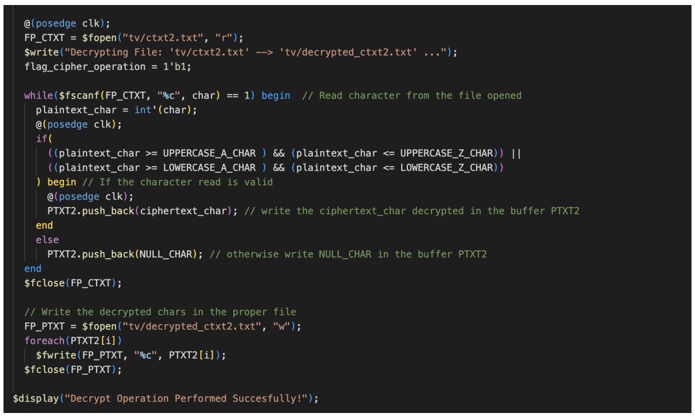
For the two ciphertext files we defined their corresponding golden models (respectively expected_ctxt1.txt and expected_ctxt2.txt files), which represent the expected output resulting out of the hardware modules processing. At this specific time instant, we compare the ciphertext produced with their respective golden model, to see if the hardware module functioned properly (fig.4.3).



```
FP_PTXT = $fopen("tv/expected_ctxt2.txt", "r");
while($fscanf(FP_PTXT, "%c", char) == 1) begin
    plaintext_char = int'(char);
    begin
  CTXT_source2.push_back(plaintext_char); // stores the read characters from file into buffer CTXT_source2
    end
  end;
  $fclose(FP_PTXT);

if(CTXT2 != CTXT_source2)begin  // Check each character encrypted has been properly stored into the file
  $display("ERROR");
  $stop;
end
  $fclose(FP_PTXT);

 $display("Compare Performed Succesfully: 'tv/ctxt2.txt' --> 'tv/expected_ctxt2.txt'");
```

Figure 4.3: Comparing Encrypted Plaintext with the Golden Model

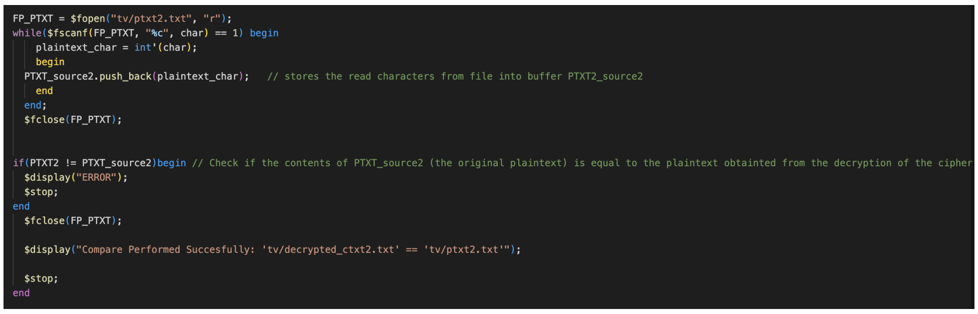
Only when the compare operation is successful, we proceed onto the inverse operation, which is the decryption operation (fig.4.4).



```
  @(posedge clk);
  FP_CTXT = $fopen("tv/ctxt2.txt", "r");
  $write("Decrypting File: 'tv/ctxt2.txt' --> 'tv/decrypted_ctxt2.txt' ...");
  flag_cipher_operation = 1'b1;

  while($fscanf(FP_CTXT, "%c", char) == 1) begin  // Read character from the file opened
    plaintext_char = int'(char);
    @(posedge clk);
    if(
      ((plaintext_char >= UPPERCASE_A_CHAR ) && (plaintext_char <= UPPERCASE_Z_CHAR)) ||
      ((plaintext_char >= LOWERCASE_A_CHAR ) && (plaintext_char <= LOWERCASE_Z_CHAR))
    ) begin // If the character read is valid
      @(posedge clk);
      PTXT2.push_back(ciphertext_char); // write the ciphertext_char decrypted in the buffer PTXT2
    end
    else
      PTXT2.push_back(NULL_CHAR); // otherwise write NULL_CHAR in the buffer PTXT2
  end
  $fclose(FP_CTXT);

  // Write the decrypted chars in the proper file
  FP_PTXT = $fopen("tv/decrypted_ctxt2.txt", "w");
  foreach(PTXT2[i])
    $fwrite(FP_PTXT, "%c", PTXT2[i]);
  $fclose(FP_PTXT);

$display("Decrypt Operation Performed Succesfully!");
```

Figure 4.4: File Decryption Operation

It's straightforward to deduce that, intrinsically with the inverse property of encryption and decryption operations, we expect to obtain the original plaintext. Once we pass each ciphertext character from the ciphertext file to the hardware module, we store the decrypted result (respectively in decrypted_ctxt1.txt and decrypted_ctxt2.txt files) and we compare them to their original plaintext (fig.4.5). Once again, if the compare is succesfull, the exit status of the file encryption tests is positive.



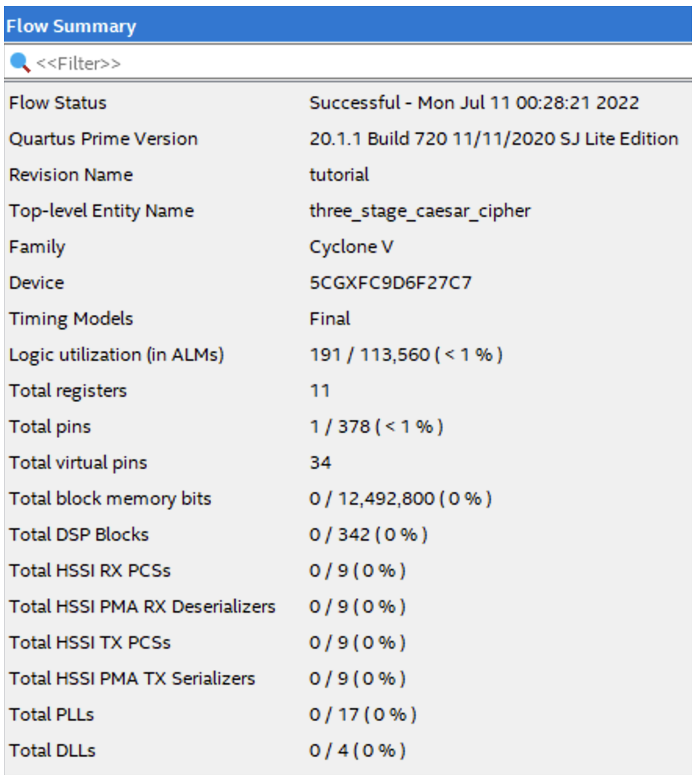
```
FP_PTXT = $fopen("tv/ptxt2.txt", "r");
while($fscanf(FP_PTXT, "%c", char) == 1) begin
    plaintext_char = int'(char);
    begin
  PTXT_source2.push_back(plaintext_char);   // stores the read characters from file into buffer PTXT2_source2
    end
  end;
  $fclose(FP_PTXT);


if(PTXT2 != PTXT_source2)begin // Check if the contents of PTXT_source2 (the original plaintext) is equal to the plaintext obtainted from the decryption of the cipher
  $display("ERROR");
  $stop;
end
  $fclose(FP_PTXT);

  $display("Compare Performed Succesfully: 'tv/decrypted_ctxt2.txt' == 'tv/ptxt2.txt'");

  $stop;
end
```

Figure 4.5: Comparing Decrypted File with Golden Model

Chapter 5

Implementation of RTL design on FPGA and results

Using Quartus it was possible to perform the phases of Circuit design, Physical design and Static Timing Analysis (STA). The chosen technology on which synthesize the module is the FPGA 5CGXFC9D6F27C7 of the Cyclone V family (Intel). The execution of the logic synthesis (Circuit design) led to the development of the NetList.


Flow Summary

<<Filter>>

Flow Status	Successful - Mon Jul 11 00:28:21 2022
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	tutorial
Top-level Entity Name	three_stage_caesar_cipher
Family	Cyclone V
Device	5CGXFC9D6F27C7
Timing Models	Final
Logic utilization (in ALMs)	191 / 113,560 (< 1 %)
Total registers	11
Total pins	1 / 378 (< 1 %)
Total virtual pins	34
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 5.1: Flow Summary

In the flow summary of the circuit (fig.5.1) there are useful information about the device, the number of registers and pins used.

The use of logic (in the ALMs) is less than 1% of the total FPGA resources. There are 11 registers:

- 8 registers for ciphertext_char
- 1 register for flag_ciphertext_ready
- 1 register for err_invalid_ptxt_char
- 1 register for err_invalid_key_shift_num

The only connected signal is the clock (1 pin), while the other registers, just mentioned, are virtual pins, which are added to:

- 8 input wires for plaintext_char
- 5 input wires for first_key_shift_number
- 5 input wires for third_key_shift_number
- 1 input wire for first_key_shift_direction
- 1 input wire for third_key_shift_direction
- 1 input wire for flag_cipher_operation
- 1 input wire for flag_valid_plaintext_char
- 1 input wire for rst_n

Some of the most relevant parts of the synthesis are shown below:

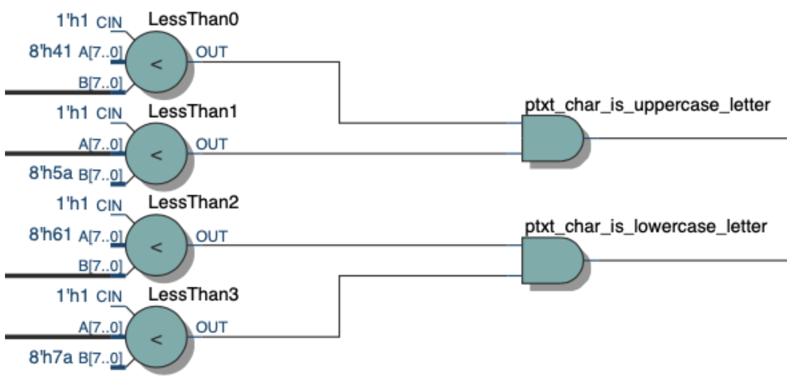


Figure 5.2: Plaintext Character Validity Check

In the figure 5.2 it is possible to observe how comparing operations are carried out, to establish whether the value is included in the interval range of uppercase letters [A-Z] or lowercase letters [a-z]. The two output wires of the two AND gates are then inputs of an OR gate (in the top part of the figure 5.3), providing the condition used for the error signals.

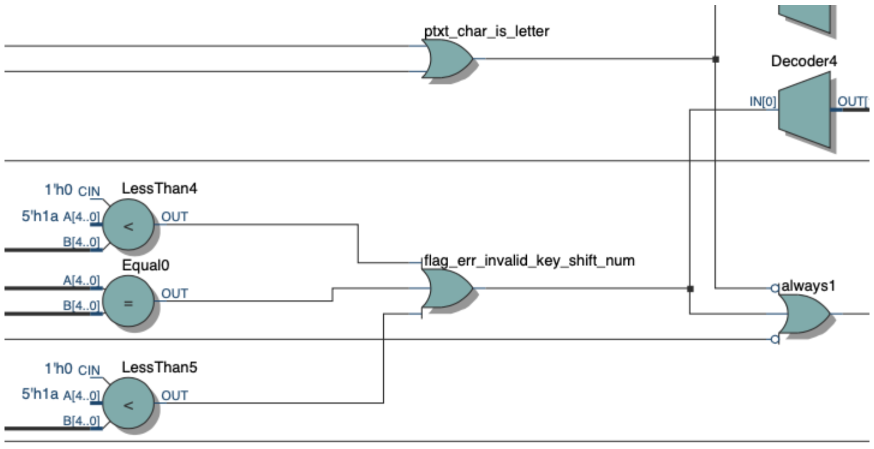


Figure 5.3: Processing of Signals that are tied to the Error Output Registers

In the figure 5.3, it is possible to observe how the Quartus tool synthesizes the HDL part related to the generation of the condition signals, which in turn are tied to the respective error outputs.

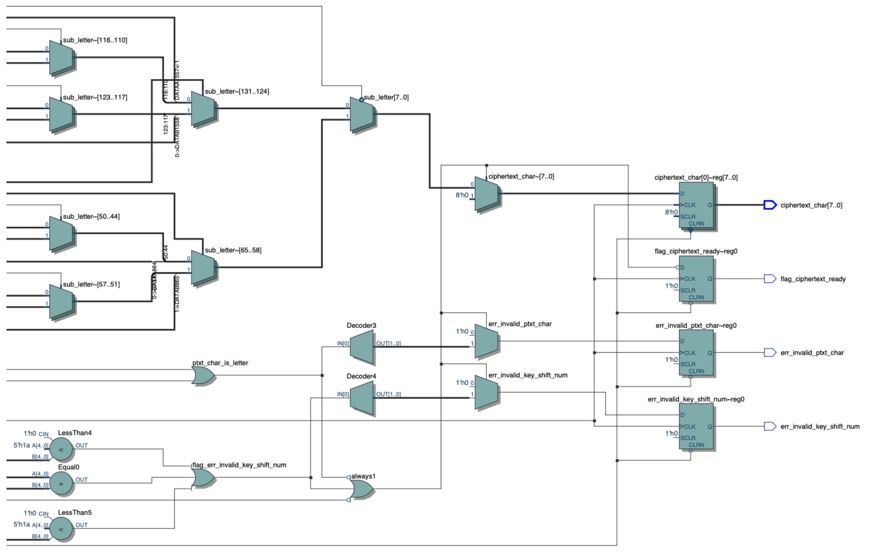


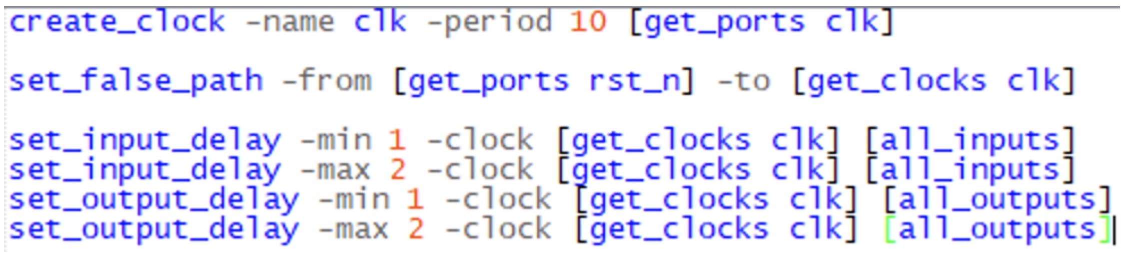
Figure 5.4: Passing the operation values and output registers

In the figure 5.4 it is possible to observe the part relating to the outputs. The last multiplexer encountered along the upper path takes care of passing the ciphertext value or the NULL value in case of an error.

Chapter 6

Static Timing Analysis (STA)

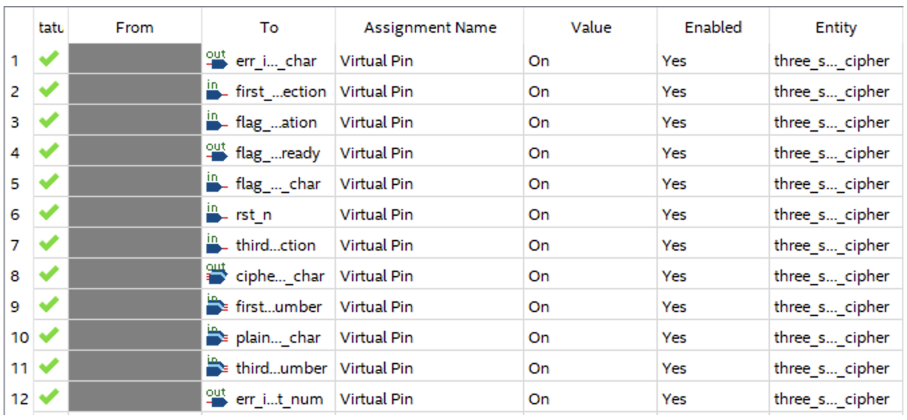
To constrain the input/output ports we have defined the timing constraint in the SDC file (fig.6.1), where each inputs and outputs are defined with max and min time delay. The delay values follow a ”rule of thumb” where it’s optimal to define them as 10% of the clock period (for the input delay) and 20% of the clock period (for the output delay).



```
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] [all_inputs]
set_input_delay -max 2 -clock [get_clocks clk] [all_inputs]
set_output_delay -min 1 -clock [get_clocks clk] [all_outputs]
set_output_delay -max 2 -clock [get_clocks clk] [all_outputs]
```

Figure 6.1: SDC file

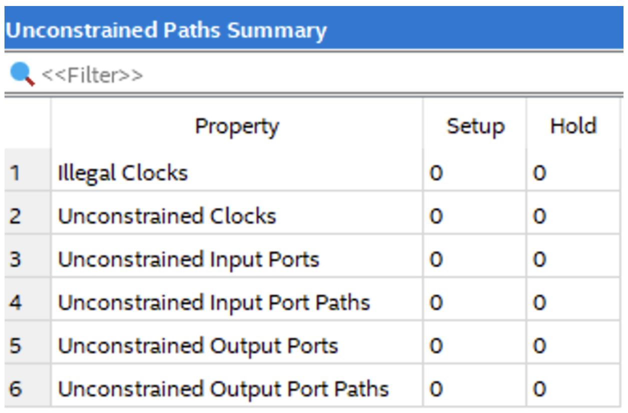
We need to assign virtual pins to increase frequency, each input and output link has its own corresponding virtual pin.



	tatu	From	To	Assignment Name	Value	Enabled	Entity
1	✓		out err_i..._char	Virtual Pin	On	Yes	three_s..._cipher
2	✓		in first_...ection	Virtual Pin	On	Yes	three_s..._cipher
3	✓		in flag_...ation	Virtual Pin	On	Yes	three_s..._cipher
4	✓		out flag_...ready	Virtual Pin	On	Yes	three_s..._cipher
5	✓		in flag_..._char	Virtual Pin	On	Yes	three_s..._cipher
6	✓		in rst_n	Virtual Pin	On	Yes	three_s..._cipher
7	✓		in third...ction	Virtual Pin	On	Yes	three_s..._cipher
8	✓		out ciphe..._char	Virtual Pin	On	Yes	three_s..._cipher
9	✓		in first...umber	Virtual Pin	On	Yes	three_s..._cipher
10	✓		in plain..._char	Virtual Pin	On	Yes	three_s..._cipher
11	✓		in third...umber	Virtual Pin	On	Yes	three_s..._cipher
12	✓		out err_i...t_num	Virtual Pin	On	Yes	three_s..._cipher

Figure 6.2: Virtual pin

Using the SDC file we have constrained all the paths in the model as we can see in the image 6.3.

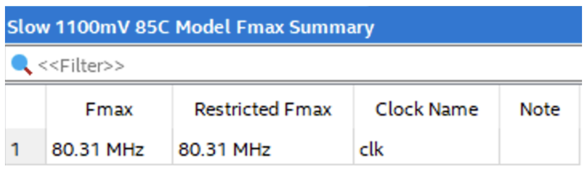


Unconstrained Paths Summary

<<Filter>>

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

Figure 6.3: Unconstrained paths

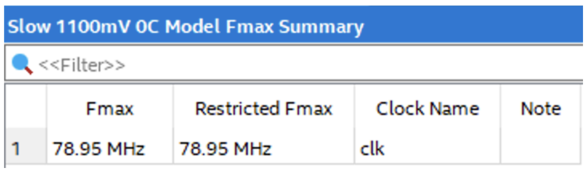


Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	80.31 MHz	80.31 MHz	clk	

Figure 6.4: Maximum frequency for corner case 1100mV 85C



Slow 1100mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	78.95 MHz	78.95 MHz	clk	

Figure 6.5: Maximum frequency for corner case 1100mV 0C

In the end, we reached a frequency of 78.95MHz in the Slow model with 0C° and 1100mV and 80.31MHz in the Slow model with 85C° and 1100mV so we didn't reach the working specification in frequency higher than 100MHz probably because we used registers which require more logic to be calculated.