

# RC4 ALGORITHM - DECRYPTION MODULE

Giuliani Andrea, Pacini Teodorico

University of Pisa  
M.Sc. Cybersecurity  
Hardware and Embedded Security



UNIVERSITÀ DI PISA

## Index of chapters

<b>1. Specification Analysis</b>	<b>3</b>
1.1 RC4 Decryption algorithm	3
1.2 Requirements	6
<b>2. Block Diagram and Design Choices</b>	<b>9</b>
<b>3. Expected Waveforms</b>	<b>12</b>
<b>4. Testbench</b>	<b>13</b>
4.1 Behavior checking	14
4.1.1 Assertion din_valid to zero	14
4.1.2 Long ciphertext	17
4.2 File decryption	17
<b>5. Implementation of RTL design on FPGA and results</b>	<b>19</b>
<b>6. Static Timing Analysis</b>	<b>21</b>
6.1 Timing Constraints File	21
6.2 Maximum Frequencies	21

# 1. Specification Analysis

## 1.1 RC4 Decryption algorithm

The project implemented a simplified version of the RC4 algorithm, a symmetric key stream cipher, for the decryption of a ciphertext. The algorithm uses a symmetric key (*KEY*) as a seed in the Key Scheduling Algorithm (*KSA*), an array (*S*) to perform a permutation mechanism and the Pseudo Random Generator Algorithm (*PRGA*) to create the keystream (*K*):

- *KSA* initializes the permutation in the *S* array, using as input the *KEY*.
- *PRGA* makes another permutation of *S* and from its values creates the *K* value, to perform the decryption of the ciphertext.

In the original implementation of the RC4 stream cipher, the key length can vary from 1 to 256 bytes, whereas following the specification of the project the size of the key should be fixed to 16 bytes.

```
for i = 0 to 255 {
    S[i] = i
}
j = 0
for i = 0 to 255 {
    j = ( j + S[i] + key[i mod 16] ) mod 256
    swap(S[i], S[j])
}
```

Figure 1: pseudo-code for Key Scheduling Algorithm

The *KSA* algorithm, respectively in *Figure 1*, is the first module of the cipher. At every cycle of the for-loop it updates the value of *j* adding to it a value of the seed and a value of *S*, in modulus 256 to avoid overflow.

The index key *i* is modulus 16 because 16 is the maximum size of the seed and by definition *i* is an eight bits signal (possible values are 0 up to 255).

Later the algorithm swaps the *i*-th element of *S* with the *j*-th element of *S*, and obtains the “permuted” *S*.

```
i = 0
j = 0
while(1){ {
    i = ( i + 1 ) mod 256
    j = ( j + S[i] ) mod 256
    swap(S[i], S[j])
    K[l] = S[(S[i] + S[j]) mod 256]
    P[l] = C[l] ⊕ K[l]
}
```

Figure 2: pseudo-code for Pseudo Random Generator Algorithm

The second module of the cipher is the *PRGA*, respectively in *Figure 2*, it handles the decryption and an additional permutation phase.

It uses two indexes ( $i, j$ ) inside a for-loop to perform the permutation on the elements of  $S$ . Subsequently, it creates the  $K$  value with different steps:

- selects  $S[i]$  and  $S[j]$
- obtains a third index  $(S[i] + S[j]) \% 256$ .
- and uses this new index to select an element inside the  $S$  array. This element will be the value of  $K$ .

All these steps are repeated in a loop a number of times equal to the size of the ciphertext.

This limitation is done for practical reasons however the cipher works as a stream cipher so it shouldn't never leave the loop and keep decrypting the incoming bytes.

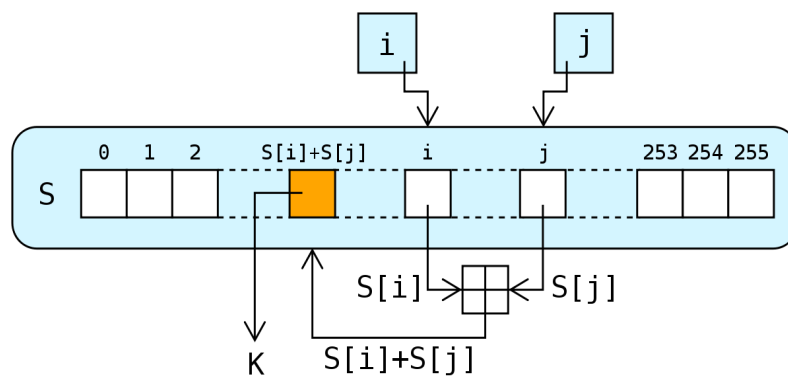


Figure 3: Keystream generation phase

## 1.2 Requirements

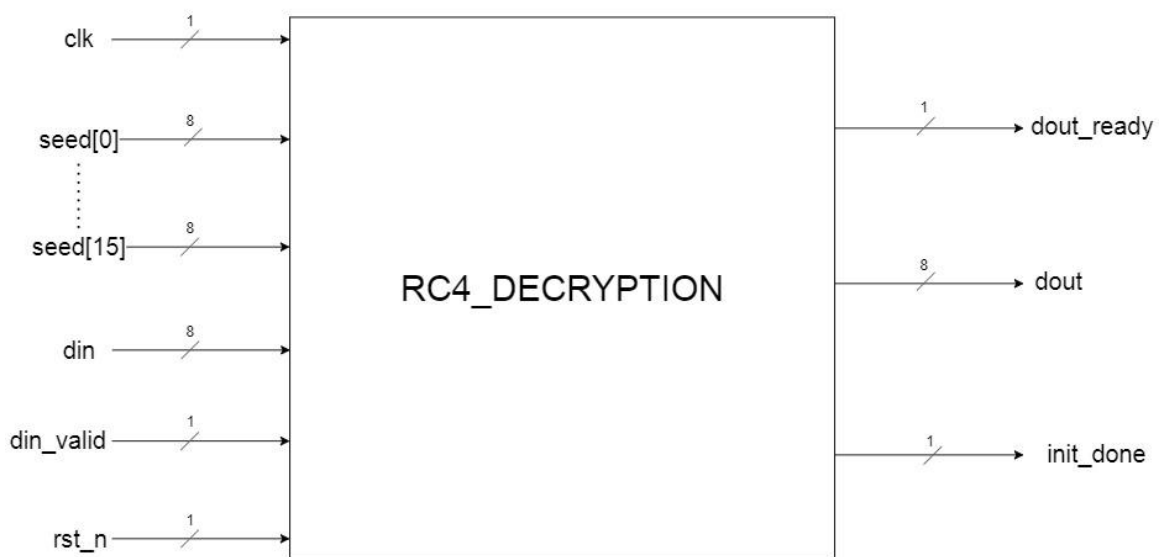


Figure 4: Implementation of RC4 decryption module

The requirements of the projects can be summarized in the following:

- 1) Given a ciphertext of arbitrary size in input, obtaining in output the corresponding plaintext (one byte at a time)
- 2) Inserting an asynchronous active-low reset port.
- 3) Adding two ports *din\_valid*, *dout\_ready* that have to be asserted when the ciphertext or the plaintext are available in input or in output. They take 1 when these values are valid and stable or 0 when these values are inconsistent.
- 4) (Optional) The decryption should give in output one byte of the plaintext for each clock cycle

About the last requirement we figured out how to do the decryption in one clock cycle, anticipating some values in the preceding clock cycles.

```
module rc4_decryption(  
    input          clk,  
    input          rst_n,  
    input  [7:0]   seed [15:0],  
    input  [7:0]   din, // ciphertext byte  
    input          din_valid,  
    output reg     dout_ready,  
    output reg [7:0] dout, // plaintext byte  
    output reg     init_done,  
);
```

*Figure 5: Decryption module input and output ports*

In the decryption module, respectively in *Figure 5*, there are the following input and output signals:

- **input clk**: clock signal.
- **input rst\_n**: asynchronous active-low reset port to give this signal without any delay related to the clock cycle.
- **input [7:0] seed [15:0]**: signals for the values of the symmetric key.
- **input [7:0] din**: signal for the ciphertext byte to decrypt when it is requested
- **input din\_valid**: signal used when the ciphertext is provided in input. If its value is 1, the corresponding input data is valid and stable. Otherwise if the value is 0, the input is inconsistent.
- **output reg dout\_ready**: signal used when the plaintext is available in the output port. If its value is 1, the corresponding data is valid and stable. Otherwise if the value is 0, the data is inconsistent.
- **output reg [7:0] dout**: signal for the plaintext byte to give in output when it is ready.
- **output reg init\_done**: signal to notify that the initialization phase has finished and the circuit is ready to decrypt.

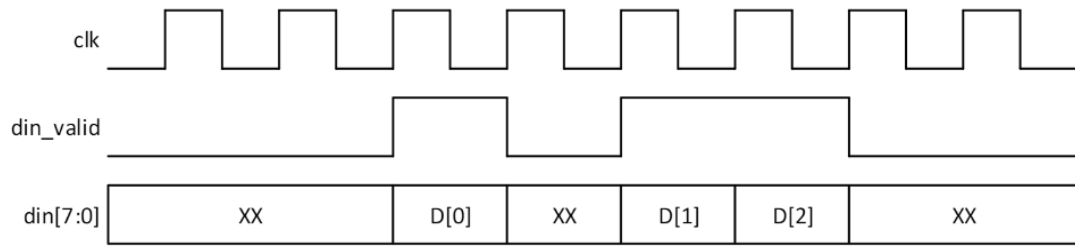


Figure 6: Example of waveform expected considering `din_valid`, `clk` and `din` signals

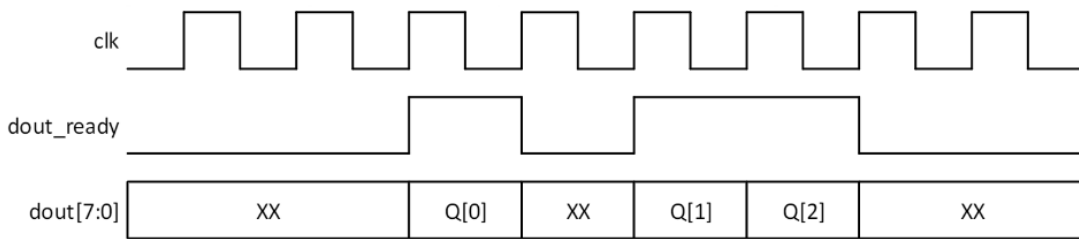


Figure 7: Example of waveform expected considering `dout_ready`, `clk` and `dout` signals

These are waveforms required by the project guidelines, respectively in *Figure 6* and *Figure 7*, showing the behaviors of signals in time considering the two situations in which we have a ciphertext valid and stable in input and a plaintext valid and stable in output. Together with the opposite situation in which the `din_valid` and `dout_ready` signals have value 0, and the `din`, `dout` signals are inconsistent.

## 2. Block Diagram and Design Choices

In this section it is shown the diagrams of the implemented RC4 decryption module, respectively the top level view and the state diagram of the finite state machine describing the behavior of the implemented algorithm.

In *Figure 8*, it is shown the top-level schematic diagram of our module with input and output signals and also internal signals. The internal signals are represented with red lines and they move internal values between sub-blocks.

The *KSA* accepts as inputs the 16 bytes of the seed and produces as output the signal *init\_done*. In addition to these it conducts operations about the first permutation of *S\_mem*, exploiting two different internal signals as inputs (*read1* and *read2*), and also two internal signals as outputs (*write1* and *write2*). These signals are used to read and to write in the *S\_mem* block, which is implemented using 256 registers.

For simplicity, in the diagram, the signals related to the writing operation on the memory are only two, basically they carry only the values to write whereas the standard implementation of the memory requires also the signals which specify the indexes of the cells in which we will write the data.

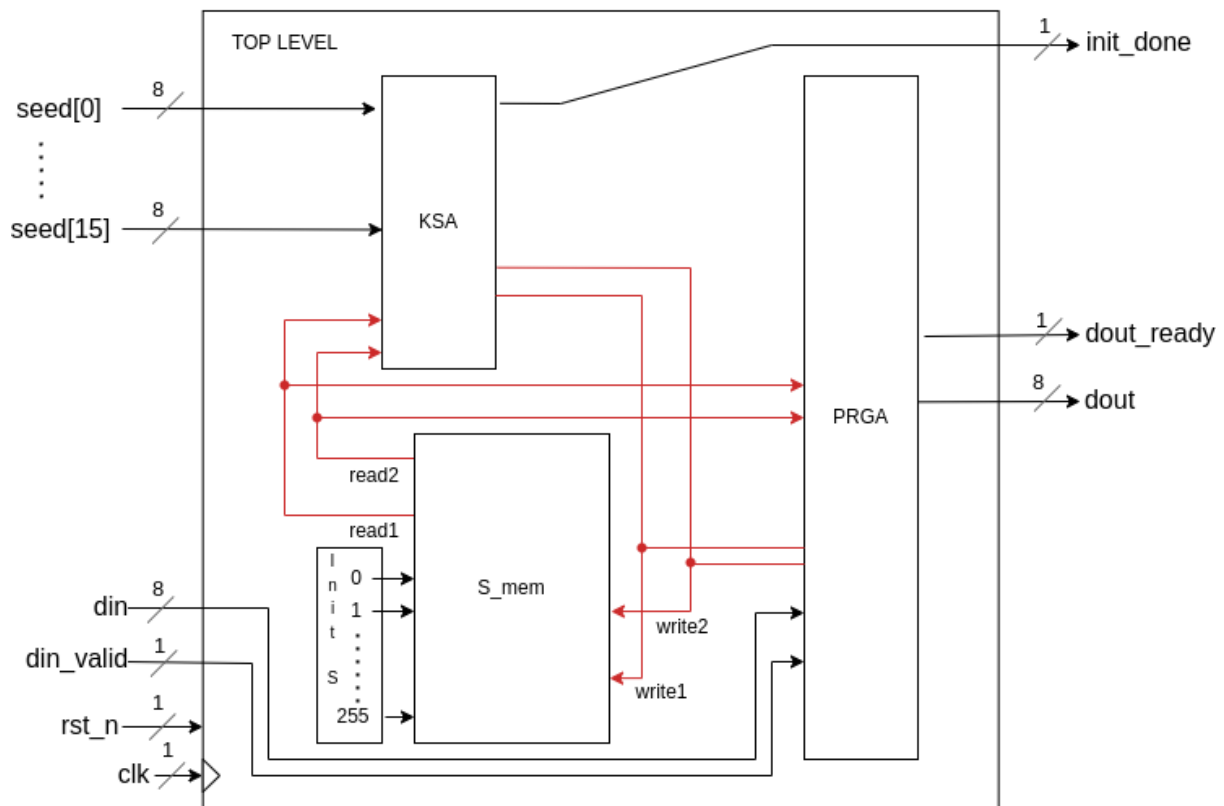


Figure 8: Top level schematic of decryption module

The *PRGA* accepts as external inputs *din* and *din\_valid* and it produces as external outputs *dout\_ready* and *dout*. As the *KSA*, it also has two internal inputs and two internal outputs to perform the read and write operations on *S\_mem*.

The *init\_S* block is used in the initial phase to assign the identity values to the 256 registers of *S\_mem*.

In *Figure 9*, we can see the state diagram of the finite state machine we have designed to control all the steps and to perform the stream decryption of the ciphertext. We have used 5 different *states* during this process:

- *RESET STATE* is used to reset the entire module during the working process or in the startup
- *STATE 0, 1, 2* are used during the *KSA* process, where the first one is used to load the identity values inside the memory *S\_mem* and to initialize the internal indexes
- *STATE 3, 4* are used during the *PRGA* process

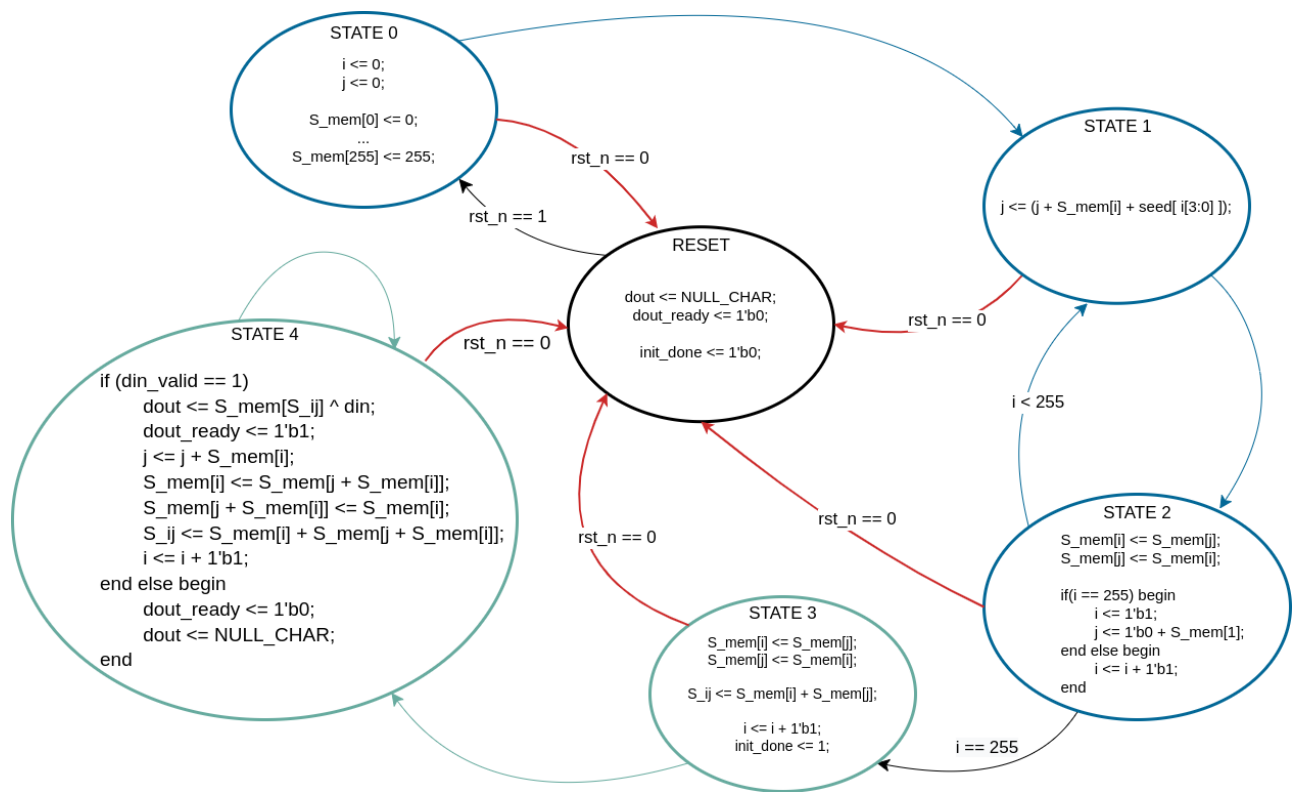


Figure 9: State Diagram of the Finite State Machine

There is no state in which *rst\_n* signal is set to 1 because it is an asynchronous input signal of the module.

Moreover, the signals in the future state maintain the previous value if not modified.

The values assigned to the signals into a state are propagated into the next state after the clock edge, therefore the condition which enables the state transition is based on the value that the signals own before the clock edge.

The finite state machine starts its execution from the **reset state**.



The *KSA* is performed by the *STATE 0*, the *STATE 1* and the *STATE 2* in blue and the *PRGA* is performed by the *STATE 3* and the *STATE 4* in green.

In the **STATE 0** the *S\_mem* array is initialized with the identity values, whereas the indexes *i* and *j* are initialized to 0.

In the **STATE 1** it is calculated the value *j*, index that will be used in the next state to access to *S\_mem*, between the addends there is a byte of the key indexed by the last four bits of *i* (key indexes go from 0 to 15).

In the **STATE 2**, it is performed the swap between two registers of *S\_mem* (*S\_mem[i]* and *S\_mem[j]*) and it is also performed a checking of the index *i* to:

- move to the next state when the permutation phase has been done in all the 256 registers of the *S\_mem* (*i* equal to 255).
- move to the previous state (*state 1*) to continue with the permutation.

The value *i* will be incremented by 1 until it reaches the value 255, at this point *i* will be set to 1. Before moving to state 3 also the value of *j* is updated, executing the second operation of the PRGA algorithm (earn a clock cycle).

During the **STATE 3**, some values are anticipated to earn a clock cycle, and to satisfy the requirement of decrypt a byte in a clock cycle, performing a swap between *S\_mem[i]* and *S\_mem[j]* and the calculation of the index *S\_ij*, the index of the value in *S\_mem* which will represent the key byte *K* ( $K = S[S_{ij}]$ ).

Subsequently, the index *i* is incremented and the *init\_done* signal is set to 1, to communicate the end of the initialization phase.

In the **STATE 4** is carried out the decryption process and the generation of the values for the next decryption. If *din\_valid* = 0, no operation will be done, we want to maintain the previous values of the signals unchanged. Only *dout\_ready* will be set to 0 and *dout* to a default value character. The decryption is not performed.

Instead, if *din\_valid* = 1, it means that the ciphertext byte in input is ready to be decrypted, we perform the XOR between the key byte and the ciphertext. Simultaneously, we update the index *j*, the value of *S\_ij* and the index of *i* is incremented for the next cycle decryption, the swap is carried out and *dout\_ready* is set to 1.

In this case the index of *S\_mem* is not *j* but  $j + S_{mem}[i]$  because otherwise we had to waste a clock cycle to sample and use the new value of *j*.

The *state* signal is always left unchanged because we will never leave this state (only in case of reset).

In the **RESET** state, the *dout* signal is set to *NULL\_CHAR* (a default value equal to 0), *dout\_ready* to 0 and the *init\_done* signal to 0. If *rst\_n* signal changes its value from 1 to 0, in whatever state the machine is located, the next state will be the reset state.

### 3. Expected Waveforms

In *Figure 12*, we can see the signal waveforms of the first “check” testbench, more specifically its behavior during the first clock cycles.

The asynchronous active-low reset signal is set to 0 to reset the circuit at startup. When the *rst\_n* signal is set to 0, the *state* is set to 0, *dout* is set to a default value (8'b0) and *dout\_ready* is set to 0.

When the *rst\_n* signal changes its value from 0 to 1, the finite state machine starts its execution from the first state, initializing the array *S\_mem* and subsequently carrying out the key scheduling algorithm.

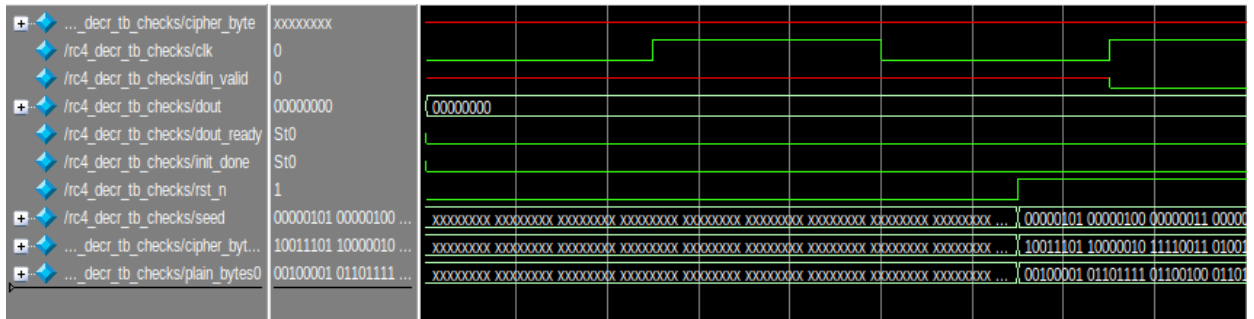


Figure 12: First part of Waveform from the first “check” testbench

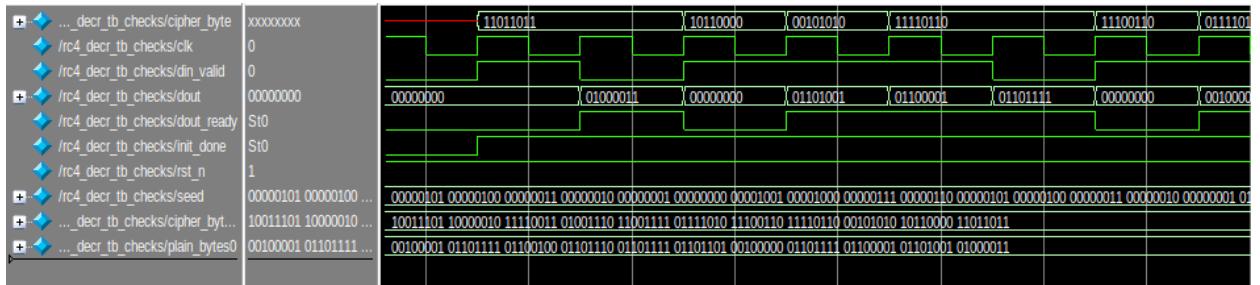


Figure 13: Second part of Waveform from the first “check” testbench

When the initialization finishes, respectively in *Figure 13*, start the decryption phase:

- If *din\_valid* is 1 the algorithm decrypts a byte of the ciphertext, then the *dout\_ready* is set to 1 and the plaintext byte is given in output with *dout*.
- If *din\_valid* is 0, the ciphertext byte will not be decrypted and the next clock cycle *dout* will be set to a default value (8'b0) with *dout\_ready* equal to 0.

## 4. Testbench

We have developed two main testbenches to test and check the main functionalities of the module. The first test is about checking the behavior of the algorithm during the decryption and inconsistent input (*din\_valid* and *dout\_ready*), whereas the second one concerns file decryption.

The test vectors have been generated with the “golden-model” algorithm implemented in Python:

- *seed0.txt*, *cipher0.txt*, *plain0.txt*, *cipher1.txt* and *plain1.txt* are used for the first testbench,
- whereas *cipher\_file.txt*, *plain\_file.txt* and *seed\_file.txt* for the file decryption.

**Note:** the python script is not inside the project folder.

```
# KSA algorithm
def key_scheduling_algorithm(key):
    # Initialization of S
    S = [i for i in range(0, 256)]

    j = 0
    for i in range(0, 256):
        # j = (j + S[i] + key[i mod 16]) mod 256
        j = (j + S[i] + to_int(key[i%16])) % 256
        # swap(S[i], S[j])
        S[i], S[j] = S[j], S[i]

    return S

# PRGA Algorithm
def encryption(S, plaintext):
    P = [ord(e) for e in plaintext]
    C = []

    i = 0
    j = 0
    for l in range(0, len(P)):
        i = (i + 1) % 256
        j = (j + S[i]) % 256
        # swap(S[i], S[j])
        S[i], S[j] = S[j], S[i]

        K = S[(S[i] + S[j]) % 256]
        C.append(P[l] ^ K)

    return C, P
```

Figure 14: Python code for KSA and PRGA algorithm

We have inserted the most relevant parts of the “golden-model”, respectively in *Figure 14*, concerning the *KSA* and *PRGA*.

The algorithm works byte-by-byte for the permutation but also for the decryption, so at every cycle it decrypts a byte of the ciphertext, moreover the script works on single ASCII

characters, especially for the encrypted file generation, it first removes the non-ASCII characters from the file and then it decrypts it.

## 4.1 Behavior checking

This testbench is made of two sub-tests generated with the *fork* instruction.

After every test, the circuit is resetted and a new ciphertext, plaintext and/or seed are loaded into the memory. We use two different registers for the ciphertexts because their dimensions are different.

### 4.1.1 Assertion *din\_valid* to zero

In all the tests of this chapter we have two blocks which run concurrently, one to handle the input and one to check the output.

```
begin: INVALID_IN_3
    for(int i = 0; i < 14; i++) begin
        if(i == 1 || i == 5 || i == 8) begin
            din_valid = 0;
            // cipher_byte = previous_value;
            @(posedge clk);

            QUEUE1.push_back(index);
            QUEUE2.push_back(din_valid);

        end else begin
            din_valid = 1;
            cipher_byte = cipher_bytes0[index];
            @(posedge clk);

            QUEUE1.push_back(index);
            QUEUE2.push_back(din_valid);
            index += 1;
        end
    end
end
end: INVALID_IN_3
```

Figure 15: Code of the first “check” testbench handling the output

At every clock cycle, the index of the character encrypted/to encrypt is pushed in a queue together with the *din\_valid* value.

```

begin: CHECK_DOUT_3
    @(posedge clk);
    $display(" #| %-9s, %-2s", "dout", "dout_ready");
    for(int i = 0; i < 14; i++) begin
        @(posedge clk);

        INDEX = QUEUE1.pop_front();
        DIN_VALID = QUEUE2.pop_front();

        if (DIN_VALID == 0) begin
            $display("%2d| %-10s %1d %1d %-5s", i, dout, dout_ready, 0, 0 == dout_ready ? "OK" : "ERROR");
            if(0 != dout_ready) begin
                $stop;
            end
        end else begin
            $display("%2d| %-1s %-1s %-5s, %1d %1d %-5s", i, dout, plain_bytes0[INDEX],
                plain_bytes0[INDEX] == dout ? "OK" : "ERROR", dout_ready, 1,
                1 == dout_ready ? "OK" : "ERROR");
            if(plain_bytes0[INDEX] != dout || 1 != dout_ready) begin
                $stop;
            end
        end
    end
end
end
end: CHECK_DOUT_3

```

Figure 16: Code of the first “check” testbench checking the output

Simultaneously (a clock cycle later), the index and the *din\_valid* value are popped from the queue by the second block:

- if *din\_valid* is 1 we check if *dout* is equal to the expected plaintext character and if *dout\_ready* is equal to 1
- otherwise if *din\_valid* is 0 we check if *dout\_ready* is equal to 0

```

# Init done!
# #| dout      , dout_ready
# 0| C C OK    , 1 1 OK
# 1|          , 0 0 OK
# 2| i i OK    , 1 1 OK
# 3| a a OK    , 1 1 OK
# 4| o o OK    , 1 1 OK
# 5|          , 0 0 OK
# 6|          , 1 1 OK
# 7| m m OK    , 1 1 OK
# 8|          , 0 0 OK
# 9| o o OK    , 1 1 OK
#10| n n OK    , 1 1 OK
#11| d d OK    , 1 1 OK
#12| o o OK    , 1 1 OK
#13| ! ! OK    , 1 1 OK

```

Figure 16: Console output of the first “check” testbench

The left column represents the *dout* value and the expected value for it, or a blank space in case *din\_valid* was zero, whereas the right column contains the obtained and expected value for *dout\_ready*.

The waveform has the behavior which we expected, when *din\_valid* is 0, at the next clock cycle no character has been decrypted and *dout\_ready* is equal to zero:

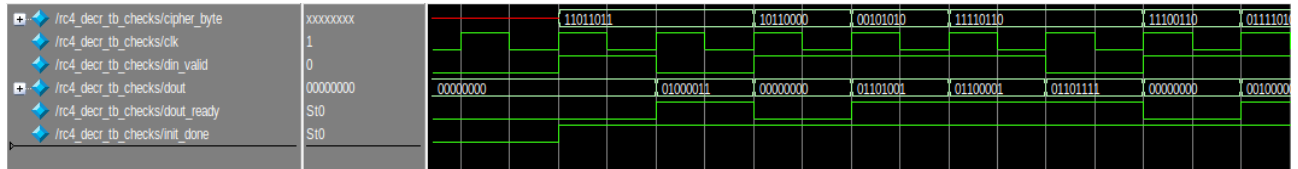


Figure 17: Waveforms of the first “check” testbench

## 4.1.2 Long ciphertext

In this test we simply input a long ciphertext, decrypt it and compare the result with the plaintext. The test is useful to observe the behavior of the algorithm when the internal indexes reach their maximum value (255).

```
begin: LONG_CIPHERTEXT_5
    index = 0;
    for(int i = 0; i < 334; i++) begin
        din_valid = 1;
        cipher_byte = cipher_bytes1[index];
        @(posedge clk);

        QUEUE1.push_back(index);
        index += 1;
    end
end: LONG_CIPHERTEXT_5

begin: CHECK_DOUT_5
    @(posedge clk);
    for(int i = 0; i < 334; i++) begin
        @(posedge clk);

        INDEX = QUEUE1.pop_front();

        if(plain_bytes1[INDEX] != dout || 1 != dout_ready) begin
            $display("%3d| %-1s %-1s %-5s, %1d %1d %-5s", i, dout, plain_bytes1[INDEX],
                plain_bytes1[INDEX] == dout ? "OK" : "ERROR", dout_ready, 1,
                1 == dout_ready ? "OK" : "ERROR");
            $display("LONG CIPHERTEXT ERROR");
            $stop;
        end
    end

    $display("LONG CIPHERTEXT OK");
end: CHECK_DOUT_5
```

Figure 18: Code of the second “check” testbench

In this test we assume that the input is always available for decryption (*din\_valid* = 1). The output will be *LONG CIPHERTEXT OK* or *LONG CIPHERTEXT ERROR* (with details of the error) in case of mismatching characters. We haven't used the same approach as before, concerning the output, because here we have three hundred characters so three hundred rows of console output are too many.

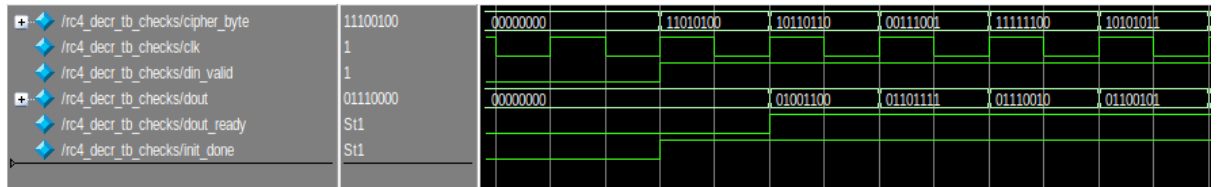


Figure 19: Part of the waveforms of the second “check” testbench

## 4.2 File decryption

The second module of the testbench file concerns the file decryption. With the Python algorithm of the RC4 stream cipher we have generated a ciphertext file from a plaintext. The test code is very intuitive: after opening the cipher file, it reads it character-by-character, every character is decrypted by the algorithm and simultaneously a second block collects the output on a queue. After the decryption, the queue is written inside the *dec.txt* file. Subsequently, we can check if *dec.txt* is equal to *plain\_file.txt*

```
begin: DECRYPT_1
    while (!$feof(FP_CTXT)) begin
        $fscanf(FP_CTXT,"%sb ", char);
        cipher_byte = char;
        din_valid = 1;
        @(posedge clk);

    end

    din_valid = 0;
    cipher_byte = NULL;
end: DECRYPT_1

begin: PUSH_PTXT_1
    @(posedge clk);
    @(posedge clk);
    while (dout_ready == 1) begin

        PTXT.push_back(dout);
        @(posedge clk);

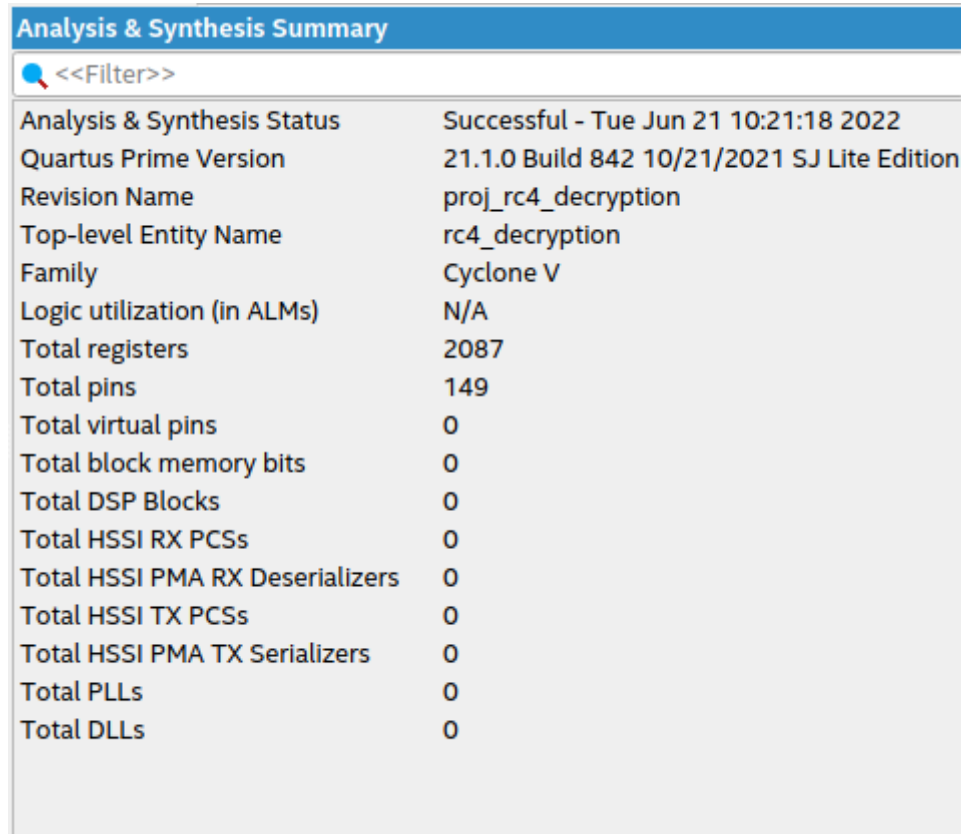
    end
end: PUSH_PTXT_1
join

$fclose(FP_CTXT);
```

Figure 21: Code of the file decryption testbench

## 5. Implementation of RTL design on FPGA and results

This chapter contains the summaries of Analysis, Synthesis and Fitter performed by Quartus, version 21.1.0. The reports, respectively in *Figure 22* and *Figure 23*, show the chosen *FPGA* of the *Cyclone V* family, in particular the device 5CGXFC9D6F27C7 and other information summarized in those figures.



Analysis & Synthesis Summary	
<<Filter>>	
Analysis & Synthesis Status	Successful - Tue Jun 21 10:21:18 2022
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	proj_rc4_decryption
Top-level Entity Name	rc4_decryption
Family	Cyclone V
Logic utilization (in ALMs)	N/A
Total registers	2087
Total pins	149
Total virtual pins	0
Total block memory bits	0
Total DSP Blocks	0
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0
Total DLLs	0

*Figure 22: Analysis and Synthesis summary from Quartus*

In *Figure 23*, we can see that the logical utilization of the *FPGA*, in other words the logical resources used in the implementation of our module, is around 5%. This result shows that this *FPGA* is not the best choice for this simple module, because there are a lot of wasted logical resources. To obtain a better solution in terms of efficiency, we could use an *FPGA* more suitable to our module, for instance a smaller *FPGA* with less resources.

The module could be used as a sub-module of a bigger device, using the pin of our module as “internal pins”, but it could be also used independently: the *clock*, the *seeds*, the *rst\_n*, the *din* and all other signals would be “external pins” which will receive some values from the external environment, to obtain an independent decryption block.




Fitter Summary	
 <<Filter>>	
Fitter Status	Successful - Tue Jun 21 10:32:49 2022
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	proj_rc4_decryption
Top-level Entity Name	rc4_decryption
Family	Cyclone V
Device	5CGXFC9D6F27C7
Timing Models	Final
Logic utilization (in ALMs)	5,361 / 113,560 ( 5 % )
Total registers	2088
Total pins	149 / 378 ( 39 % )
Total virtual pins	0
Total block memory bits	0 / 12,492,800 ( 0 % )
Total RAM Blocks	0 / 1,220 ( 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 23: Fitter summary from Quartus

## 6. Static Timing Analysis

The *Static Timing Analysis (STA)* is a method of validating the timing performance of a design by checking all possible paths for timing violations, using an .sdc file to provide the required timing constraints.

### 6.1 Timing Constraints File

```
create_clock -name clk -period 10 [get_ports clk]

# get_clocks takes the name specified with "-name" and not the one of the port
set_false_path -from [get_ports rst_n] -to [get_clocks clk]

# min. 1 ns of input and output delay max. 2ns
# all_input automatically exclude the clock
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 24: Timing Constraints File

In Figure 24, it is shown the .sdc file to perform the static timing analysis.

The first command `create_clock` has the role to create a clock object with a period of 10ns, and it is linked with the input signals `clk` in the module.

The command `set_false_path` defines what signals are outside the timing analysis. In our case we do not consider the `rst_n` signal because it has an asynchronous behavior, without any relationships with the clock signal.

The commands `set_input_delay` and `set_output_delay` define port delay for input and output signals. For both input and output ports there is a minimum and maximum delay: the minimum delay has been set to 1 ns and the maximum delay to 2 ns.

### 6.2 Maximum Frequencies


Slow 1100mV 85C Model Fmax Summary				
 <<Filter>>				
	Fmax	Restricted Fmax	Clock Name	Note
1	71.15 MHz	71.15 MHz	clk	

Figure 25: Slow 1.1V 85°C model max frequency


Slow 1100mV 0C Model Fmax Summary				
 <<Filter>>				
	Fmax	Restricted Fmax	Clock Name	Note
1	71.73 MHz	71.73 MHz	clk	

Figure 26: Slow 1.1V 0°C model max frequency

From *STA* we see the maximum frequency supported by our implementation, on two different models on which our circuit is tested.

With a temperature of 85°C, respectively in *Figure 25*, the maximum frequency is equal to 71.15MHz.

With a temperature of 0°C, respectively in *Figure 26*, the maximum frequency is equal to 71.73MHz. It has to be considered the lower frequency of 71.15MHz because this is the worst case in which our module is able to work.