

Hardware and Embedded Security
Project - Simplified RC4 stream cipher (encryption module)
Master in Cybersecurity - 2021/2022
University of Pisa

Marc Hofmann
Francesco Rubino

Date of delivery: 13.07.2022

Table of contents

Table of contents	2
Specification Analysis	3
Introduction	3
Specifications	3
Requirements	4
Block diagram and design choices (RTL design module)	6
Explanation of code	6
Schematic	13
RTL analysis from Quartus	14
Expected waveforms	18
Modelsim Waveforms	18
Beginning of process	18
Encryption	19
Conclusion	20
Testbench	20
Architectural and Functional Description of testbench	20
Verification methodology / approach	23
Implementation of RTL design on FPGA and results	24
Analysis & Synthesis Summary	24
Fitter Summary	25
Static Timing Analysis	26
SDC file	26
Virtual Pin Assignment	26
Frequency results	26
Unconstrained Paths Summary	27
List of Figures	28
List of Sources	29

1. Specification Analysis

a. Introduction

RC4 is an extremely simple to use Stream Cipher. It was developed by Ron Rivest of RSA Security in 1987 and is used in popular protocols like WEP, Skype, Remote Desktop Protocol and optionally even in TLS / SSL. Sadly due to several design flaws, RC4 cannot be considered to be secure anymore. There have been a variety of potential attack vectors reported. Due to these weaknesses WEP for example had to be retired and replaced by (now also outdated and insecure) WAP. The 128-Bit key of RC4 can - for example - be cracked within a short time. Therefore RC4 is not commonly used today anymore (1)(2)(3).

In this project, a simplified version of RC4 will be implemented as part of the Hardware and Embedded Security class at the University of Pisa.

b. Specifications

Simplified Version of RC4

As explained in the description, the RC4 implementation will be simpler in our project compared to the original RC4 specification by Prof. Ron Rivest.

Normally in RC4, the key length can vary between 1 - 256 bits. In our case, the key length will be fixed at 128-bits, or in other words 16 bytes.

This can be represented by :

$$j = (j + S[i] + key[i \bmod 16]) \bmod 256$$

Subroutines

To make the encryption work, two subroutines will be implemented. There is the Key Scheduling Algorithm followed by the Pseudo-Random Generation Algorithm. Both are described on the next page.

Key Scheduling Algorithm

In the case of RC4 a Key Scheduling algorithm is used to initialize the permutation in the array S. The key length, as explained above, is fixed at 128 bits. The array S is initialized to the identity permutation. S is then processed for 256 iterations in a similar way to the main PRGA, but mixes in bytes of the key at the same time.

Below is the Pseudo-C code provided in the project description, to better understand and explain the KSA.

```

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])
}

```

Pseudo-Random Generation Algorithm

Here, PRGA modifies the state and outputs a byte of the keystream. In each iteration, PRGA first increments i . Then PRGA looks up the i th element of S , which is $S[i]$ and adds that to j . It then exchanges the values of $S[i]$ and $S[j]$ and then uses sum $S[i] + S[j] \bmod 256$ as an index to fetch a third value of S . Afterwards, XOR takes place, with the next byte of the message to produce the next byte of the ciphertext or plaintext.

This produces a stream of $K[0], K[1], \dots$ which are XORed with the plaintext to obtain the ciphertext. In other words $C[l] = P[l] \oplus K[l]$.

As taken from the description of the project, below is Pseudo-C code to describe the PRGA:

```

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]
    C[l] = P[l] ⊕ K[l]
}

```

c. Requirements

There are some additional requirements that are to be implemented.

One byte at a time

The above described PRGA and KSA-Algorithm shall be used for one byte of plaintext at a time. This then outputs one byte of corresponding ciphertext.

Clock Cycle

The stream cipher shall encrypt one plaintext byte per clock cycle. In other words, PRGA and KSA-Algorithm should conduct one clock cycle for each byte of plaintext that is to be encrypted.

Asynchronous active-low reset port

The RC4 stream cipher shall have an asynchronous active-low reset port. This port permits to lead the module to the initial state in the case we would use a different key for a new encryption.

Input Port Specification

The stream cipher shall feature an input port which has to be asserted when providing an input plaintext byte (din_valid port): 1'b1, when input byte is valid and stable, 1'b0, otherwise; the following waveform is expected at input interface of stream cipher.

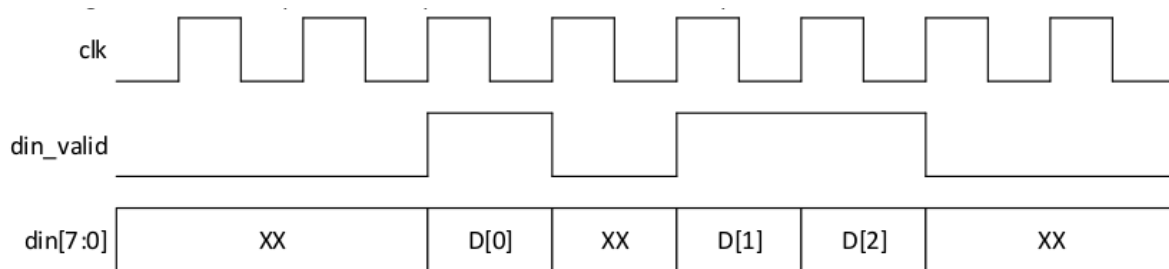


Figure 1: Expected waveform specification as described in the project description - input port

Output Port Specification

The stream cipher shall feature an output port which is asserted when the generated output ciphertext byte is available at the corresponding output port (dout_ready port): 1'b1, when output ciphertext byte is valid and stable, 1'b0, otherwise; this flag shall be kept to logic 1 at most for one clock cycle; the following waveform is expected at the output interface of stream cipher.

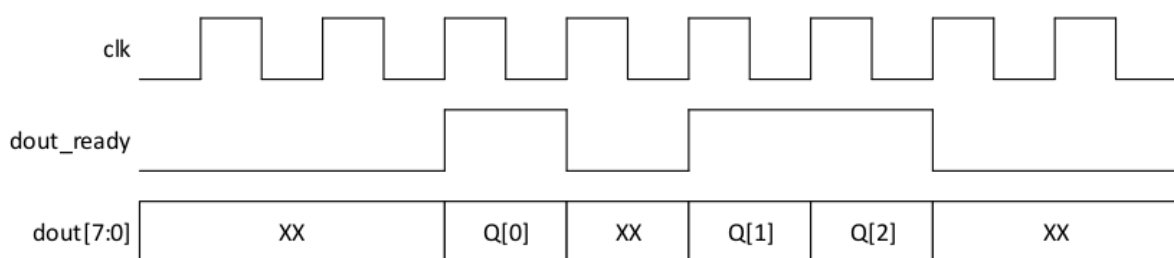


Figure 2: Expected waveform specification as described in the project description - output port

Computation of $K[i]$

Perform computation of $K[i]$ in advance with respect to the clock cycle in which $P[i]$ is provided as input by means of port $\text{din}[7:0]$.

For sources on the specification part see project assignment (4).

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

a. Explanation of code

```
localparam NULL_CHAR = 8'h00;

module RC4encryption
[
    input clock,
    input rst_n,
    input [127:0] key,
    input valid_key,
    input [7:0] plaintext,
    input valid_din,
    output reg ready_for_key,
    output reg ready_for_plaintext,
    output reg [7:0] ciphertext,
    output reg valid_dout
];
```

Figure 3: Declaration of input and output pins

At the beginning of our design, we declare one localparam NULL_CHAR with a length of 8.

The module RC4 encryption itself is then started with declaring several inputs and output registers (ports). They are explained below:

- **input clock:** Serves as our clock signal.
- **input rst_n:** This is the asynchronous active-low reset signal.
- **input [127:0] key:** Is our key, at a fixed length of 128 bits, as explained in the requirements above
- **input valid_key:** Serves as a signal input to determine if the key is valid, or not

- **input [7:0] plaintext:** Is a signal of fixed length of 8 bits, as we encrypt one byte at a time
- **input valid_din:** Serves as a signal input to determine if the input plaintext is valid
- **output reg ready_for_key:** Serves as a signal output to determine if at this stage of the process, the key can be accepted
- **output reg ready_for_plaintext:** Serves as a output to determine if at this state of the process, the plaintext can be accepted
- **output reg ciphertext [7:0]:** Is a register that outputs the ciphertext. It is 8 bits long, as we encrypt one byte at a time
- **output_reg valid_dout:** Serves as a signal to determine if the output of the ciphertext is valid

```
reg [7:0] permutation_box [0:255];
reg [7:0] i;
reg [7:0] j;
reg [7:0] l;
reg [127:0] used_key;
reg [2:0] status;
```

Figure 4: Declaration of several other supporting registers

- **reg [7:0] permutation_box [0:255]:** Is a central aspect of this design. A permutation box cluster is declared. This has a length of 256 permutation boxes. There needs to be 256 permutation boxes, as we conduct 256 permutations for the key scheduling algorithm. Each of the permutation boxes can take 8 bits as a value. It is a data structure that contains all the numbers from 0 to 255. Depending on the key given at input, these numbers are shuffled and the box is initialized. The keystream used for the encryption is generated from operations made on the permutation_box.
- **reg [7:0] i, j:** These registers serve as counters and indexes used during the operations of the algorithms (KGA and PRGA).
- **reg [7:0] l:** This register is used as the index of the permutation box that is used for the next encryption.
- **reg [127:0] used_key:** Stores our key and is later used as a support signal.
- **reg [2:0] status:** Is 3 bits long and later is used to determine cases of the key scheduling and the encryption. Later its functionality becomes clearer.

Lastly, several parameters are created (figure 5) that later serve to determine the cases that are currently required to perform the operations to key schedule and encrypt. They are used in the status register.

```
parameter S0 = 0;
parameter S1 = 1;
parameter S2_1 = 2;
parameter S2_2 = 3;
parameter S3 = 4;
parameter S4 = 5;
parameter S5 = 6;
parameter S6 = 7;
```

Figure 5: Declaration of case parameters

At this stage, the actual initialization of the process begins.

```
always @(posedge clock or negedge rst_n)
begin
if(!rst_n)
begin
    ready_for_key <= 1'b0;
    ready_for_plaintext <= 1'b0;
    ciphertext <= NULL_CHAR;
    valid_dout <= 1'b0;
    status <= S0;
end
else
begin
    case(status)
    S0:
    begin
        permutation_box[0] <= 8'b00000000;
        permutation_box[1] <= 8'b00000001;
```

Figure 6: First block of functional code - establishing settings and starting case S0

This code block starts with declaring clock and resets. We use “always @ (posedge clock or negedge rst_n)”, so that at every positive edge of the clock or at the negative edge of the rst_n, the entire following code block will be executed. If the asynchronous active-low reset signal is 0, we set ready_for_key, ready_for_plaintext and valid_dout to 0. We also set the status to S0, to begin the first status / case.


```

permutation_box[253] <= 8'b11111101;
permutation_box[254] <= 8'b11111110;
permutation_box[255] <= 8'b11111111;
status <= S1;
ready_for_key <= 1'b1;
ready_for_plaintext <= 1'b0;
end

```

Figure 7: End of first block of functional code - case S0

In this S0 case we fill the 256 permutation boxes with the numbers from 0 to 255. These numbers will be shuffled later during the key scheduling algorithm execution, making it ready for the encryption of plaintext bytes. All permutation boxes are being filled at the same time. Once this has happened and the positive edge clock occurs, the status register switches to begin the next case. In addition, read_for_key is set to 1 as we now are able to receive and use the key to conduct key scheduling. ready_for_plaintext is set to 0 at this point, as it is currently not possible to encrypt any plaintext.

```

S1:
begin
  if (valid_key)
  begin
    used_key <= key;
    status <= S2_1;
    i <= 0;
    ready_for_key <= 0;
    j <= 0;
  end
  else
  begin
    ready_for_key <= 1'b1;
    used_key <= 0;
    status <= S1;
  end
end
end

```

Figure 8: Block of functional code - case S1

Status S1 is now active. If the key is valid, used_key is equaled to our key. At the same time, the status is switched. i is set to 0, just like ready_for_key and j.

On the other hand, if the key is not valid, ready_for_key remains at 1, while the used_key is reset, as it is not valid. The status is S1 again, to do this step all over again.

Now that we have the key, the module should properly initialize the permutation box executing the operations on the second part of the KSA. These operations are made in the steps S2_1 and S2_2 and are repeated 256 times each.

```

S2_1:
begin
    j <= j + permutation_box[i] + used_key[i[3:0]*8+:8];
    status <= S2_2;
end

```

Figure 9: Block of functional code - case S2_1

In Step 2_1 a calculation is conducted to determine the value of j. This operation is equal to the C-Pseudocode $j = (j + S[i] + key[i \bmod 16]) \bmod 256$ of the key scheduling algorithm.

```

S2_2:
begin
    permutation_box[i] <= permutation_box[j];
    permutation_box[j] <= permutation_box[i];
    if (i == 255)
    begin
        status <= S3;
    end
    else
    begin
        status <= S2_1;
        i <= i+1'b1;
    end
end

```

Figure 10: Block of functional code - case S2_2

Status S2_2 is now active. Here we do our main part of the key scheduling which can be compared to the $swap(S[i], S[j])$ operation of the C-Pseudocode. Permutation box i gets swapped with permutation box j, and vice versa. Once it has reached 256 permutations, the status is changed to proceed to the next step, the actual encryption.

```

S3:
begin
    i <= 1;
    j <= permutation_box[1];
    status <= S4;
end

```

Figure 11: Block of functional code - case S3

In S3 i is set to 1, while the value of j is set to the value of permutation_box[1]. This step emulates the first part of the first iteration of the PRGA, updating the reg i and j with the values that they should have before the swap operation.

```

S4:
begin
    //SWAP
    permutation_box[i] <= permutation_box[j];
    permutation_box[j] <= permutation_box[i];

    // This can be done because:  $S[i] + S[j] =$ 
    1 <= permutation_box[i] + permutation_box[
    ready_for_plaintext <= 1'b1;

    // the variables for the next round can al
    i <= i+1'b1;
    if (j == i+1'b1)
    begin
        j <= j + permutation_box[i];
    end
    else
    begin
        j <= j + permutation_box[i+1'b1];
    end

    status <= S5;
end

```

Figure 12: Block of functional code - case S4

In Status S4 the swapping of the permutation_boxes takes place. This is comparable to the C-Pseudocode $swap(S[i], S[j])$. The index of the permutation box used for the first encryption (the register i) is then filled with the sum of the permutation_boxes i and j (we can do this thanks to the commutative property), while ready_for_plaintext is set to 1 as it is now able to receive the next values of the plaintext.

Of course to use the indexes i and j for the operations of the algorithm we should set them properly, occupying clocks for the correct calculation of them. Anyway we want to create a module that is able to give at output one ciphertext per clock. This means that we must be able to encrypt one block of one byte per clock. To get this feature we should calculate some values in advance and infer the values that some registers will have later. This would allow us to use the values of the register i and j sooner. Some operations of status S4 and the entire status S5 are completely based on this principle, achieving the property we talked about. In particular, beyond the operations described before, in the status S4 i and j are already initialized with the values needed for the next iteration, despite we are still working on the first one waiting for the plaintext in input.

The condition on j is fundamental: to properly set j we assume that the next iteration 'i' will be incremented by 1 so we can use 'permutation_box[i+1]'. This works always but in the case "j == i+1". In fact, in this case, permutation_box[i+1] (that is permutation_box[j] right now) will be swapped with permutation_box[i] at the end of the clock interval and this means that it is inconsistent now. Anyway, if we know that it will be swapped with permutation_box[i], we can use it for the calculation of j to bypass this limitation.

```

S5:
begin
  if(valid_din)
  begin
    ciphertext <= permutation_box[l] ^ plaintext;
    valid_dout <= 1'b1;

    //i and j have been set properly in the status S4
    l <= permutation_box[i] + permutation_box[j];

    // We set i and j properly for the next calculation of l and
    // for the swap
    i <= i+1'b1;
    if (j == i+1'b1)
    begin
      j <= j + permutation_box[i];
    end
    else
    begin
      j <= j + permutation_box[i+1'b1];
    end

    // Now I make the swap operation
    permutation_box[i] <= permutation_box[j];
    permutation_box[j] <= permutation_box[i];

    status <= S5;
  end
  else
  begin
    status <= S5;
    valid_dout <= 1'b0;
  end
end
end

```

Figure 13: Block of functional code - case S5

At the status S5 the values provided by the status S4 are the index i and j already set for the swap of the next iteration and the index l of the key used for the encryption of the current iteration.

If we obtain a valid plaintext at input:

- we give at output the ciphertext obtained by the xor of the plaintext and $\text{permutation_box}[l]$
- we set the valid_out to 1 to keep available the ciphertext
- we calculate the index l for the next encryption. This is possible thanks to the initialization of i and j on the previous status and to the commutative property (even if it is going to have a swap in this stage the calculation of the index l is still correct).
- we set i and j as status S4.
- we swap $\text{permutation_box}[i]$ and $\text{permutation_box}[j]$

In the case of a non valid input we just reset the reg valid_out to be sure that the ciphertext is not considered as a good one. Once the module has reached the status S5, it will remain on it until a reset signal or the shutdown of the system occurs, thanks to the fact that all the operations needed for the encryption are made in just one clock cycle.

```

end
default:
begin
    status <= S0;
end
endcase
end
end
endmodule

```

Figure 14: Block of functional code - ending

The last noteworthy aspect in this module is the default. In case of any problems, the default is the status S0, e.g. we return to the declaration of the permutation boxes. The process starts over again, from the beginning.

b. Schematic

Below is a very simple, high level schematic of the circuit (figure 15). It shows relevant inputs, outputs and the KSA as well as the encryption module.

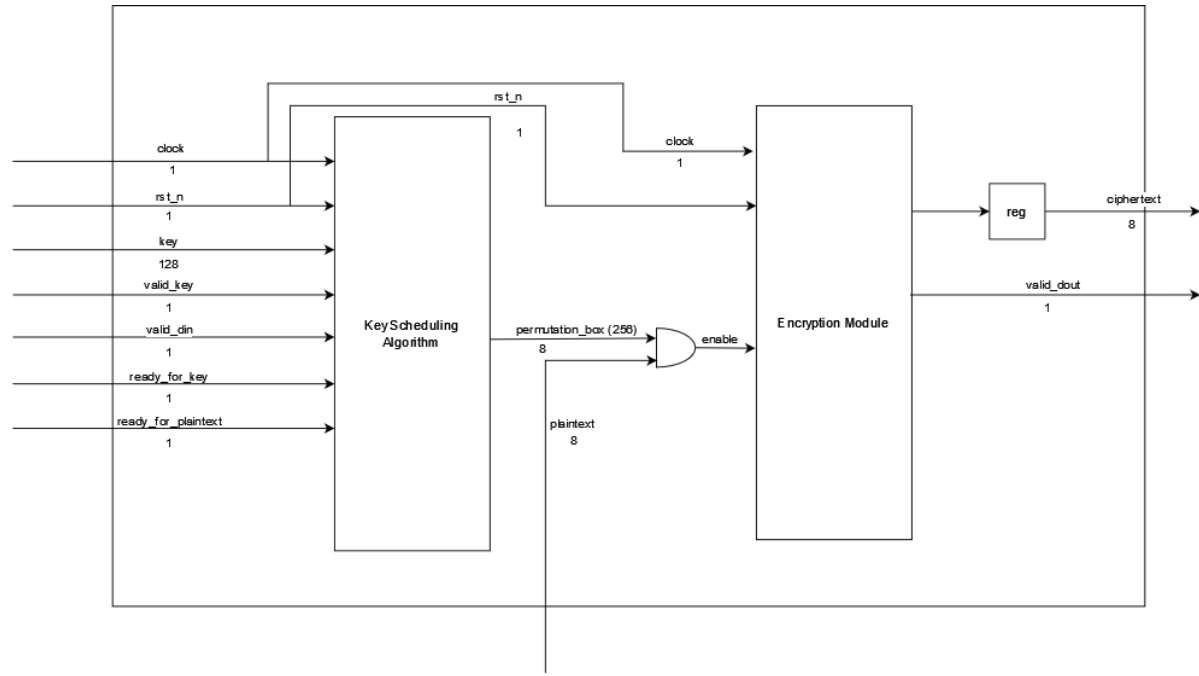


Figure 15: High level schematic of the design

c. RTL analysis from Quartus

Due to the large number of registers in the circuit, the RTL viewer and report itself is rather difficult to read and report here. Nevertheless, some selected screenshots (figure 16, 17, 18, 19, 20) can be seen below, which give an overview of the components defined in the code above.

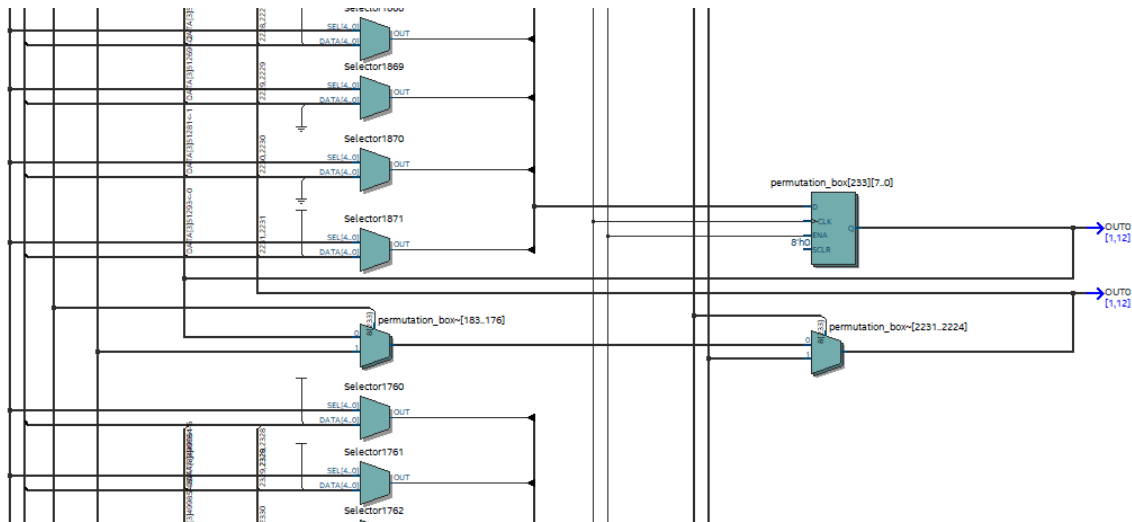


Figure 16: RTL viewer results 1

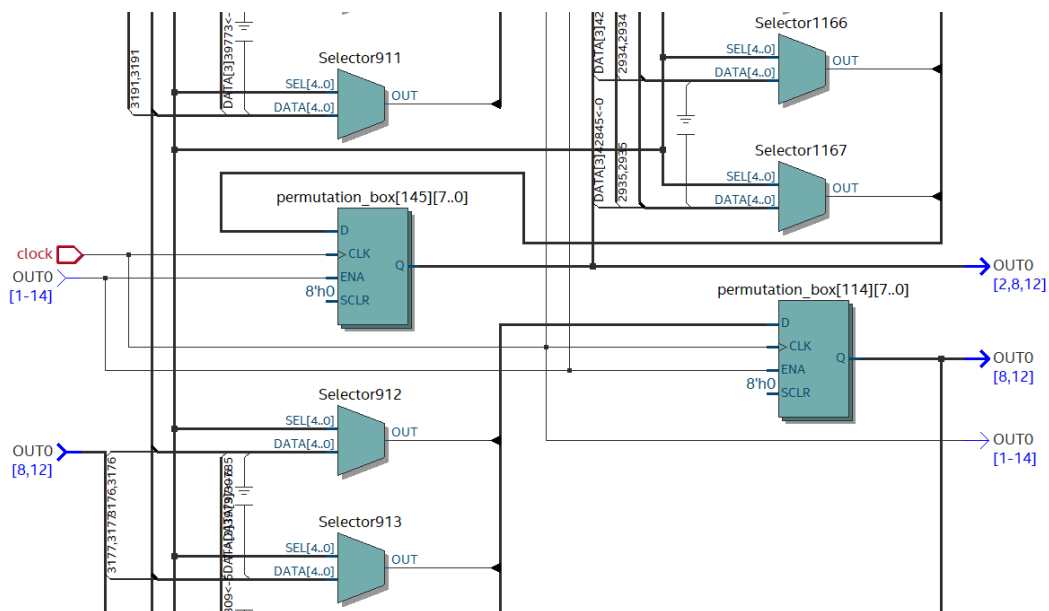


Figure 17: RTL viewer results 2

The first screenshot (figure 16) is an exemplary screenshot that shows permutation_boxes and several selectors. Most of the circuit looks relatively close to this, or the second

screenshot (figure 17) just below. Figure 17 also contains the clock input, seen on the left side.

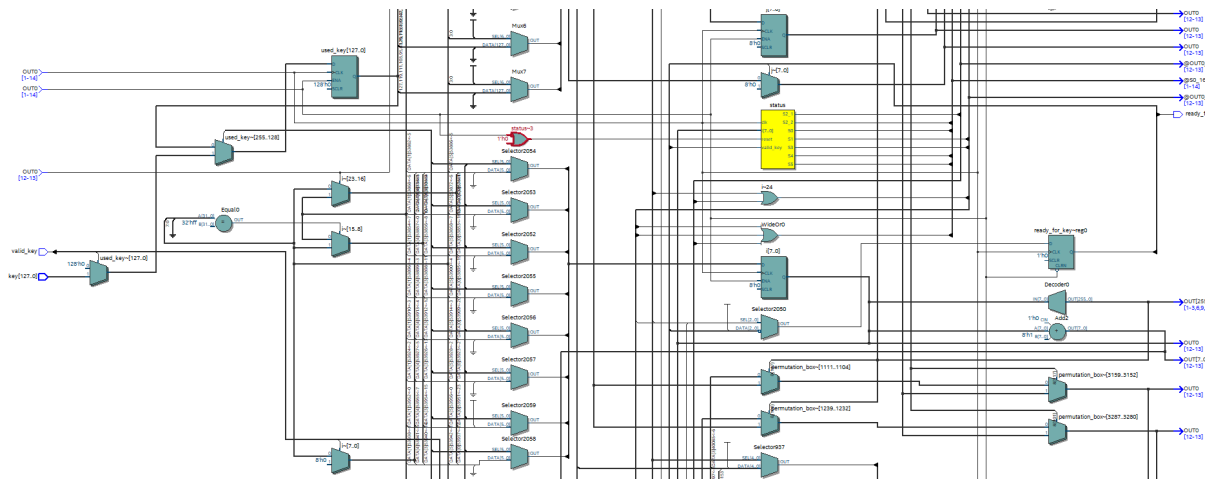


Figure 18: RTL viewer results 3

Figure 18 shows our status register (yellow).

It also shows several inputs like the [0...127] key or valid_key. An output that is visible is ready_for_key.

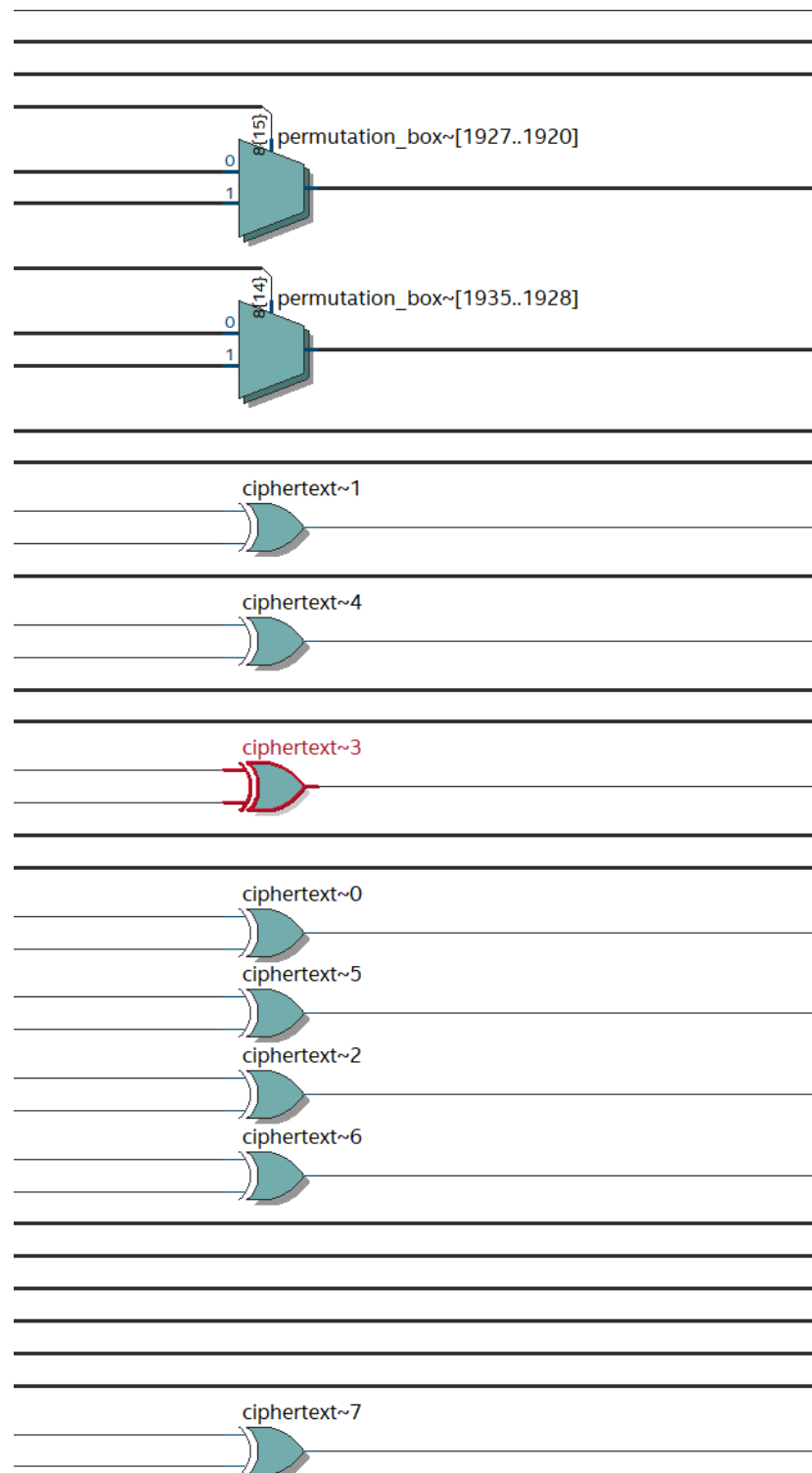


Figure 19: RTL viewer results 4

Figure 19 gives an overview of 2 permutation boxes, but more importantly the 8 bits of ciphertext that serve as the output.

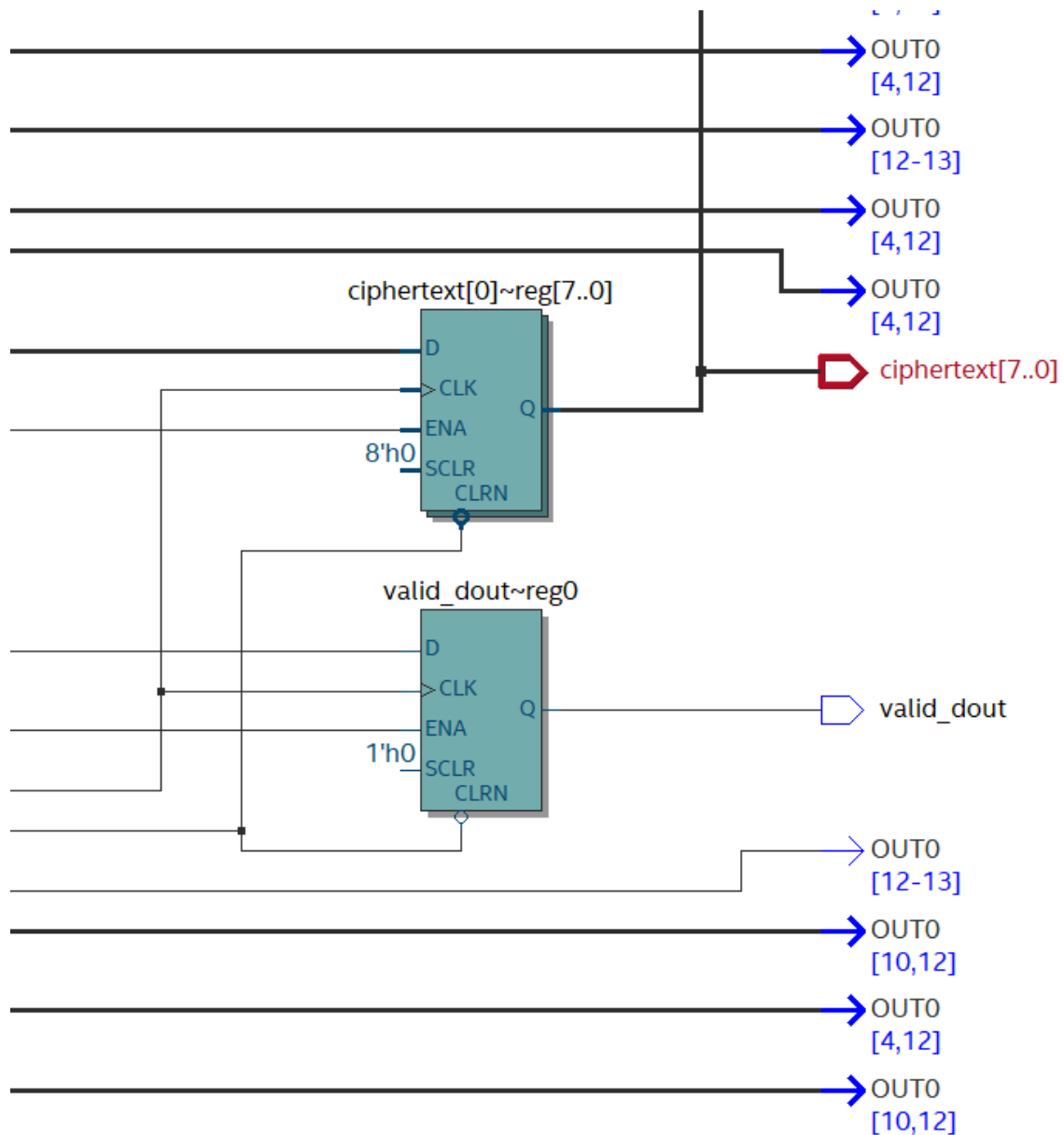


Figure 20: RTL viewer results 5

In the last screenshot (figure 20) we see 2 important things. One output is the ciphertext. Its corresponding ciphertext register can also be seen. The valid_dout register and its output valid_dout are also visible.

3. Expected waveforms

a. Modelsim Waveforms

i. Beginning of process

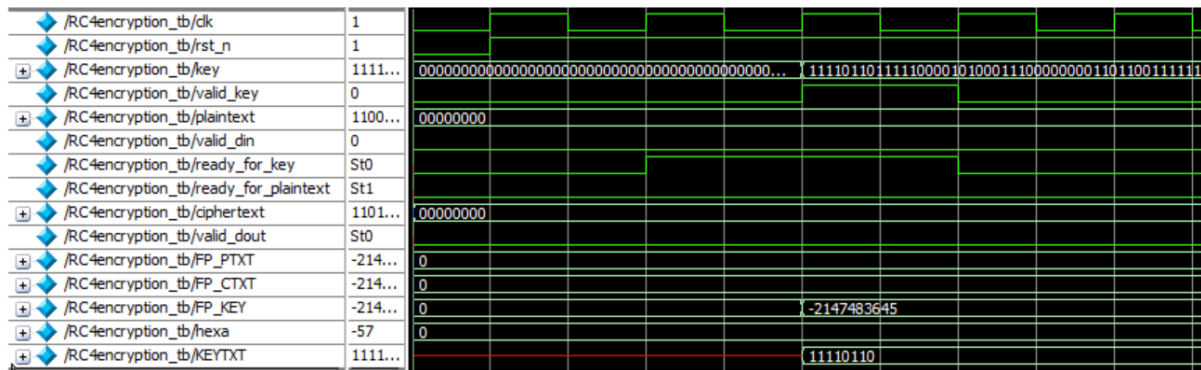


Figure 21: Modelsim waveform - beginning of encryption process

- **clk**: This screenshot shows the beginning of the encryption process. We can clearly see the clock, with a rhythmic wave form that stretches over the entire process.
- **rst_n**: rst_n gets activated with the first clock cycle and remains this way until the end.
- **key**: The key gets loaded at the beginning. It is filled with padding. For the first 2 clock cycles, the only thing that happens here is the validity of the key being confirmed. Once that confirmation is accepted, the valid key flag gets set to 1.
- **valid_key**: Gets set to 1, and remains so for 1 clock cycle, as only 1 confirmation is needed and not a continuous one.
- **plaintext**: At first the plaintext gets loaded, only with 00000000. Changes to this later are explained below.
- **valid_din**: At this stage, valid_din is not affected.
- **ready_for_key**: Ready for key is set to 1 for a period to two clock cycles. Setting ready_for_key = 1, happens before valid_key is confirmed.
- **ready_for_plaintext**: At this stage, ready for plaintext is not affected.
- **ciphertext**: At this point, the value of ciphertext is set to 00000000, as there has not happened any ciphertext calculation yet.
- **valid_dout**: At this point, valid_dout is not affected, as there is no output to be shown yet.

ii. Encryption

At this point, the actual interesting thing, the encryption takes place. The waves have at this point changed to the following:

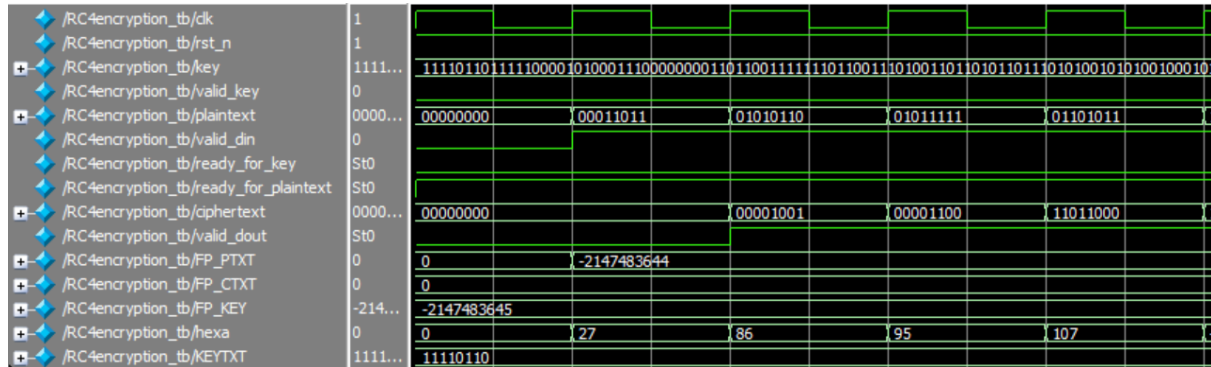


Figure 22: Modelsim waveform - encryption and end of process

- **clk**: The clock remains the same waveform as above.
- **rst_n**: Remains the same waveform as above.
- **key**: The key here is still filled with the values that were opened above. In our case, the key is read from a file and its value is cc28bec716a9d4ad4d677f36c051f8f6.
- **valid_key**: The key has been determined to be valid before, and therefore is at 0 here, as this confirmation is not needed anymore.
- **plaintext**: At this point the plaintext is read, one byte per clock cycle. In our case, the plaintext is read from a file which says “1b565f6bce1bde2f9e5363c7” (Hex). The first byte (00011011) read here corresponds to “1b”, the second (01010110) to “56” and so on. It ends with 11000111, which corresponds to the final “c7” in “1b565f6bce1bde2f9e5363c7”
- **valid_din**: is set to 1 and remains set to 1 until the last plaintext byte has been read.
- **ready_for_key**: Is set to 0, as the key has been read before and is not needed anymore.
- **ready_for_plaintext**: Is set to 1, once the process is accepting the plaintext to encrypt. It stays at 1, even after the last byte of plaintext has been read.
- **ciphertext**: The ciphertext output is generated once the first plaintext byte has been read.
- **valid_dout**: is set to 1, until the encryption process is over.

iii. Conclusion

Overall, we can determine that the waveforms are in line with the project requirements.

If we take a look at the waveform of the clock, and the plaintext, we can determine that 1 byte of plaintext is encrypted, per clock cycle. Therefore the optional requirement (see above) appears to be fulfilled.

4. Testbench

a. Architectural and Functional Description of testbench

The very simple testbench aims to practically verify the functionality of the RC4 encryption. Due to this we begin declaring a register `clk`, set at 0 (figure 23). It is set at 0 because an initial value is assigned, without this the module doesn't know what value the clock has at the beginning.

```
module RC4encryption_tb;
    reg clk = 1'b0;
    always #5 clk = !clk;

    reg rst_n = 1'b0;

    initial begin
        @(posedge clk) rst_n = 1'b1;
    end
end
```

Figure 23: Code block of testbench - part 1

```
reg [127:0] key = 128'b0;
reg valid_key = 1'b0;
reg [7:0] plaintext = 8'b0;
reg valid_din = 1'b0;

wire ready_for_key;
wire ready_for_plaintext;
wire [7:0] ciphertext;
wire valid_dout;

RC4encryption INSTANCE_NAME (
    .clock                (clk)
    ,.rst_n               (rst_n)
    ,.key                 (key)
    ,.valid_key           (valid_key)
    ,.plaintext           (plaintext)
    ,.valid_din           (valid_din)
    ,.ready_for_key       (ready_for_key)
    ,.ready_for_plaintext (ready_for_plaintext)
    ,.ciphertext          (ciphertext)
    ,.valid_dout          (valid_dout)
);
```

Figure 24: Code block of testbench - part 2

In the next step, several registers and wires are declared (figure 24). Those include - among others - key, plaintext, valid_din and valid_dout. In addition, several instance names are declared that serve as an interface to the main function of the encryption algorithm.

```
int FP_PTXT;  
int FP_CTXT;  
int FP_KEY;  
byte hexa;  
reg [7:0] CTXT [$];  
reg [7:0] PTXT [$];  
reg [7:0] KEYTXT;
```

Figure 25: Code block of testbench - part 3

Additionally, int FP_PTXT, int FP_CTXT, int FP_KEY and reg [7:0] CTXT [\$] and reg [7:0] PTXT [\$] are declared. Additionally one byte called hexa is declared. Those functions later help us to store the value of the files to be used, and also to iterate through those stored values.

At this point, we enter the main testbench functionality.

```
initial begin  
    @(posedge clk); // INITIALIZATION PERMUTATION BOX (S0)  
    @(posedge clk); // UNTIL READY_FOR KEY IS SET UP  
  
    while (ready_for_key == 1'b0)  
    begin  
        @(posedge clk);  
    end  
  
    key = 128'b0;  
    FP_KEY = $fopen("tv/key.txt", "r");  
    while($fscanf(FP_KEY, "%2h", KEYTXT) == 1)  
    begin  
        key = {KEYTXT, key[127:8]};  
    end  
  
    valid_key = 1'b1;  
    @(posedge clk);  
    valid_key = 1'b0;
```

Figure 26: Code block of testbench - part 4

Firstly in figure 26, we start two clock cycles, to initialize the permutation box, and secondly to wait until the ready_for_key register is set up.

While `ready_for_key` is still at 0, new clock cycles are begun on a constant basis. Once `ready_for_key` is set to 1, we open the text file that contains the key with the “`$fopen`” command. The key gets read from the file using `$fscanf`, and the ‘key’ register is properly initialized. At this point, `valid_key` is set to 1, as a valid key is present. A further clock cycle is added. Lastly, since the key has been read at this point, `valid_key` is set to 0 again.

```
while (ready_for_plaintext == 1'b0)
begin
    @ (posedge clk);
end

FP_PTXT = $fopen("tv/plaintext.txt", "r");
$display("Encrypting file 'tv/plaintext.txt' to 'tv/ctxt.txt'...");
```

Figure 27: Code block of testbench - part 5

A somewhat similar mechanism that was described for the key above, happens now for the plaintext (figure 27). Until `ready_for_plaintext` is set to 1, a clock cycle is pushed and the relevant files get opened.

```
while($fscanf(FP_PTXT, "%2h", hexa) == 1)
begin
    plaintext = byte'(hexa);
    valid_din = 1'b1;
    @ (posedge clk);
    if (valid_dout == 1'b1)
    begin
        CTEXT.push_back(ciphertext);
    end
end
valid_din = 1'b0;
@ (posedge clk);
```

Figure 28: Code block of testbench - part 6

At this point, we start actually encrypting the plaintext. “`while($fscanf(FP_PTXT, "%2h", hexa) == 1)`”, reads and determines if there is still plaintext to encrypt. If there is, `valid_din` is set to 1 and a clock is passed. If there is a valid output, then the ciphertext is pushed back. The “push back” operation consists of inserting the ciphertext at the end of the queue “CTXT”. When there is no more plaintext to encrypt, `valid_din` = 0 and one clock cycle is passed (figure 28).

```

        FP_CTXT = $fopen("tv/ciphertext.txt", "w");
        foreach (CTXT[i]) $fwrite(FP_CTXT, "%2h", CTXT[i]);
        $fclose(FP_CTXT);

        $display("Done!");
        @ (posedge clk);
        $stop;
    end
endmodule

```

Figure 29: Code block of testbench - part 7

We now open the ciphertext.txt file. In this file we write the output of the calculated ciphertext. As a last step, one clock cycle is passed. The module ends at this point (figure 29).

b. Verification methodology / approach

Due to simplicity reasons and a lean code, we decided to use a manual approach to verifying the functionality of our circuit. We used the test vectors provided with the project assignment to verify the correct functionality of the project.

These 5 test vectors contain key, plaintext and the expected ciphertext of the output (all in Hexadecimal). All of them vary in length, and all of them have been determined to be functional according to our tests. Using these test vectors, gives us confidence that our RC4 encryption is working as intended. Below is one example, on how we verified.

1. Step: Fill key.txt with the provided key (cc28bec716a9d4ad4d677f36c051f8f6)
2. Step: Fill plaintext.txt with the provided plaintext (1b565f6bce1bde2f9e5363c7)
3. Step: Run the simulation.
4. Step: Open the newly created ciphertext.txt file.
5. Step: Compare this ciphertext with the provided ciphertext by Prof. Crocetti. They are equal, which confirms the correctness of the simulation.

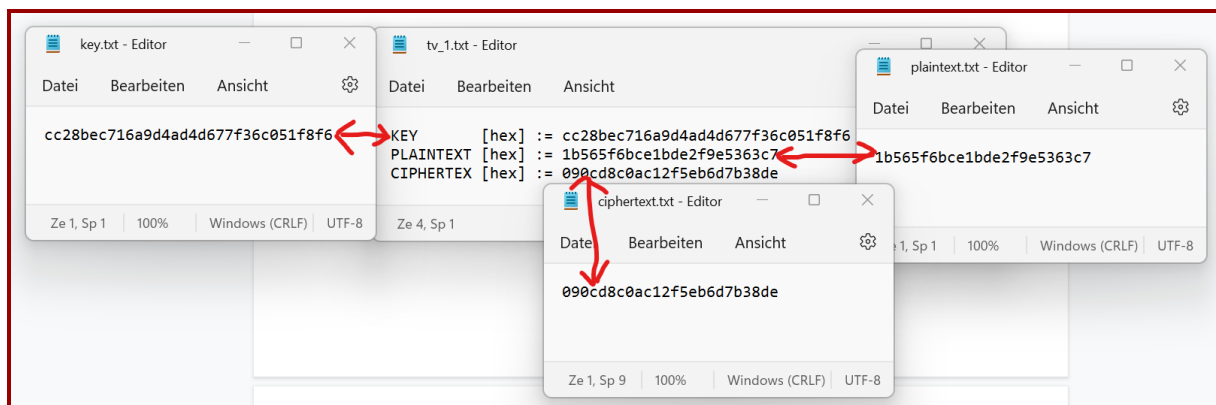


Figure 30: Results of encryption test - verification

The same process was applied to the 4 other files. All of them worked as expected and therefore correctly.

5. Implementation of RTL design on FPGA and results

The following steps are all successful by Quartus. No error messages or warnings are thrown (except the allowed ones specified by the assignment).

a. Analysis & Synthesis Summary

Analysis & Synthesis Summary	
<<Filter>>	
Analysis & Synthesis Status	Successful - Tue May 31 22:44:55 2022
Quartus Prime Version	20.1.0 Build 711 06/05/2020 SJ Lite Edition
Revision Name	proj_RC4encryption
Top-level Entity Name	RC4encryption
Family	Cyclone V
Logic utilization (in ALMs)	N/A
Total registers	2218
Total pins	1
Total virtual pins	150
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 31: Screenshot of Analysis & Synthesis Summary of Quartus

As can be seen in the Analysis & Synthesis summary (figure 31), 150 virtual pins are assigned. The relatively high number of pins comes from the high dimension of the key input used for the configuration of the permutation boxes. 1 pin is a physical pin, the clock.

We expected a huge amount of registers. In total we use 2218 registers, this larger number comes due to the nature of the permutation boxes.

b. Fitter Summary

Fitter Summary	
<<Filter>>	
Fitter Status	Successful - Tue May 31 22:48:51 2022
Quartus Prime Version	20.1.0 Build 711 06/05/2020 SJ Lite Edition
Revision Name	proj_RC4encryption
Top-level Entity Name	RC4encryption
Family	Cyclone V
Device	5CGXFC9D6F27C7
Timing Models	Final
Logic utilization (in ALMs)	3,505 / 113,560 (3 %)
Total registers	2220
Total pins	1 / 378 (< 1 %)
Total virtual pins	150
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 32: Screenshot of Fitter Summary Quartus

As can be seen in the figure 32, around 3% of the logic utilization is used. Only 1 of 378 potential pins is used, as the rest of pins are virtual. According to the fitter, there are 2220 total registers.

6. Static Timing Analysis

a. SDC file

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

set_false_path -from [get_ports rst_n] -to [get_clocks clock]

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

Figure 33: Screenshot of SDC file

The SDC file (figure 33) has been kept very simple, but as the following frequencies in the screenshots show, fulfills its purpose perfectly.

b. Virtual Pin Assignment










	tatu	From	To	Assignment Name	Value	Enabled	Entity
1	✓		 ready_for_plaintext	Virtual Pin	On	Yes	RC4e...tion
2	✓		 rst_n	Virtual Pin	On	Yes	RC4e...tion
3	✓		 valid_din	Virtual Pin	On	Yes	RC4e...tion
4	✓		 valid_dout	Virtual Pin	On	Yes	RC4e...tion
5	✓		 valid_key	Virtual Pin	On	Yes	RC4e...tion
6	✓		 ciphertext	Virtual Pin	On	Yes	RC4e...tion
7	✓		 key	Virtual Pin	On	Yes	RC4e...tion
8	✓		 plaintext	Virtual Pin	On	Yes	RC4e...tion
9	✓		 ready_for_key	Virtual Pin	On	Yes	RC4e...tion
10		<<new>>	<<new>>	<<new>>			

Figure 34: Screenshot of assignment of virtual pins Quartus

Virtual Pins have been assigned to all components, except the clock. This of course improved the frequency by a considerable margin.

c. Frequency results

The maximum Frequency is 100.95 MHz. All timing constraints are fulfilled and no warnings or errors are displayed by Quartus after conducting the timing analysis.

The design is quicker on the 85C Model, by about 0,05 MHz.

Slow 1100mV 0C Model Fmax Summary			
<<Filter>>			
	Fmax	Restricted Fmax	Clock Name
1	100.95 MHz	100.95 MHz	clock

Figure 35: Screenshot frequency result Slow 1100mV 0C model Quartus

Model	FMax
Slow 1100mV 85C Model	101.00 MHz
Slow 1100mV 0C Model	100.95 MHz

Table 1: Frequency results both models

d. Unconstrained Paths Summary

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 36: Quartus summary of unconstrained paths

As can be seen in figure 36, there are no unconstrained paths in the design, which is of course what we were aiming for.

List of Figures

Figure 1: Expected waveform specification as described in the project description - input port

Figure 2: Expected waveform specification as described in the project description - output port

Figure 3: Declaration of input and output pins

Figure 4: Declaration of several other supporting registers

Figure 5: Declaration of case parameters

Figure 6: First block of functional code - establishing settings and starting case S0

Figure 7: End of first block of functional code - case S0

Figure 8: Block of functional code - case S1

Figure 9: Block of functional code - case S2_1

Figure 10: Block of functional code - case S2_2

Figure 11: Block of functional code - case S3

Figure 12: Block of functional code - case S4

Figure 13: Block of functional code - case S5

Figure 14: Block of functional code - ending

Figure 15: High level schematic of the design

Figure 16: RTL viewer results 1

Figure 17: RTL viewer results 2

Figure 18: RTL viewer results 3

Figure 19: RTL viewer results 4

Figure 20: RTL viewer results 5

Figure 21: Modelsim waveform - beginning of encryption process

Figure 22: Modelsim waveform - encryption and end of process

Figure 23: Code block of testbench - part 1

Figure 24: Code block of testbench - part 2

Figure 25: Code block of testbench - part 3

Figure 26: Code block of testbench - part 4

Figure 27: Code block of testbench - part 5

Figure 28: Code block of testbench - part 6

Figure 29: Code block of testbench - part 7

Figure 30: Results of encryption test - verification

Figure 31: Screenshot of Analysis & Synthesis Summary of Quartus

Figure 32: Screenshot of Fitter Summary Quartus

Figure 33: Screenshot of SDC file

Figure 34: Screenshot of assignment of virtual pins Quartus

Figure 35: Screenshot frequency result Slow 1100mV 0C model Quartus

Figure 36: Quartus summary of unconstrained paths

List of Sources

- (1) <https://crashtest-security.com/disable-ssl-rc4/>
- (2) <https://www.techtarget.com/searchsecurity/answer/Should-the-RC4-cipher-still-be-used-in-enterprises>
- (3) <https://www.encryptionconsulting.com/education-center/what-is-rc4/>
- (4) See project assignment