Hardware and Embedded Security

Project - Simplified RC4 stream cipher (encryption module)

Marc Hofmann

Francesco Rubino

Master in Cybersecurity

University of Pisa

1. **Specification Analysis** 
   1. Introduction

RC4 is an extremely simple to use Stream Cipher. It is fast and therefore has a high performance and can also be used by large datasets that are to be encrypted.

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 Due to these weaknesses WEP for example had to be retired and replaced by (now also outdated and insecure) WAP.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 seconds. Therefore RC4 is not commonly used today anymore.

Nevertheless, for an introduction to hardware designing, RC4 presents several advantages, like the aforementioned simple implementation. 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.

* 1. 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 order words 16 bytes.

This can be represented by :

𝑗 = ( 𝑗 + 𝑆[𝑖] + 𝑘𝑒𝑦[𝑖 𝑚𝑜𝑑 16] ) 𝑚𝑜𝑑 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]* mod 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]*, … which are XORed with the plaintext to obtain the ciphertext. In other words C[*l*] = P[*l*] ⊕ 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]

}

* 1. 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

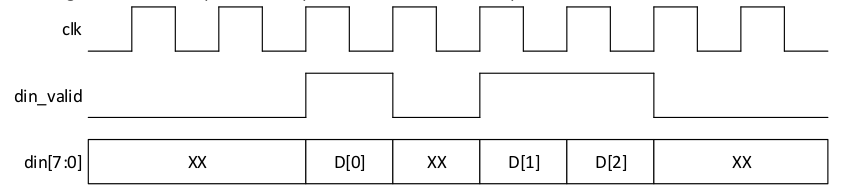
The stream cipher shall encrypt one plaintext byte per clock cycle. In other words, PRGA and KSA-Algorithm should conduct one clock cycle for each byte of plantext that is to be encrypted.

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.

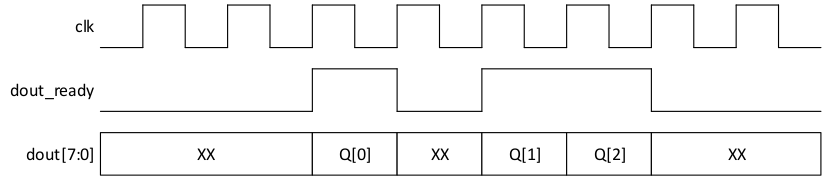
Input Port Specification

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.



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.

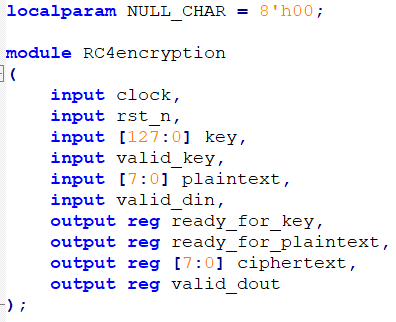


Computation of *K[l]*

Perform computation of *K[l]* in advance with respect to the clock cycle in which *𝑃[𝑙]*

is provided as input by means of port din[7:0].

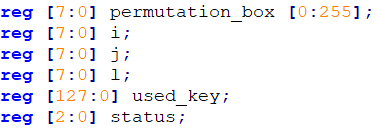
1. **Block diagram and design choices (RTL design module)** 
   1. Explanation of code



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

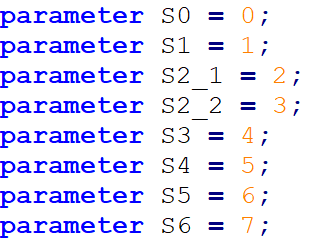
The module RC4 encryption itself is then started with declaring several inputs and output registers (ports). They are explained below:

* **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\_plainext:** 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

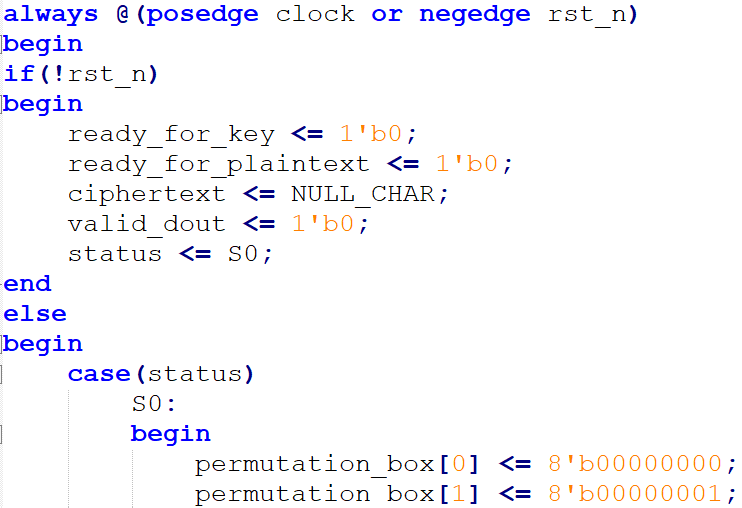
At the next step, several registers are created. 

* **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 properly initialized.
* **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.

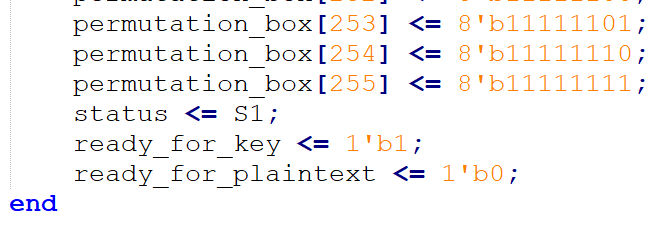
Lastly, several parameters are created that later serve to determine the cases that are currently required to perform the operations to key schedule and encrypt. They are used in the status register.



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

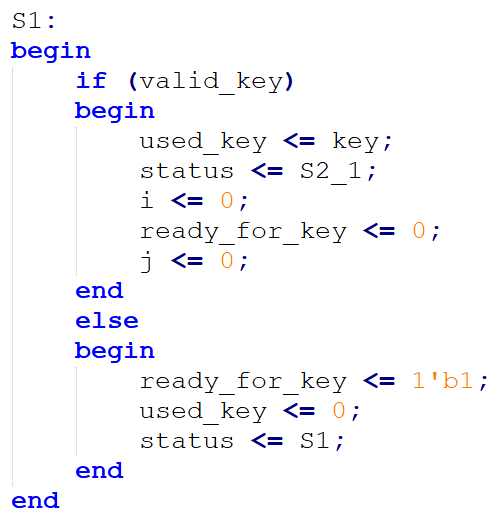


This code block starts with declaring clock and rests. We use “always @ (posedge clock or negedge rst\_n)”, so that at every positive edge of the clock or at the negative edge of the rst\_n, the entire following code block will be executed. If the asynchronous active-low reset signal is 0, we set ready\_for\_key, ready\_for\_plaintext and valid\_dout to 0. We also set the status to S0, to begin the first status / case.



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.

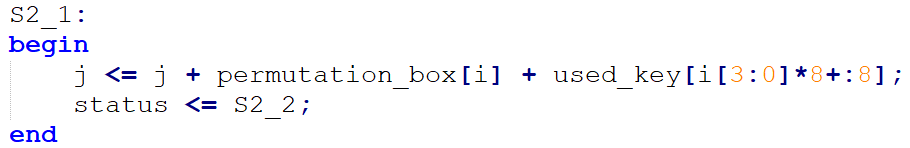
Once this has happened and the positive edge clock occurs, the status register switches to begin the next case. In addition, read\_for\_key is set to 1 as we now are able to receive and use the key to conduct key scheduling. ready\_for\_plaintext is set to 0 at this point, as it is currently not possible to encrypt any plaintext.

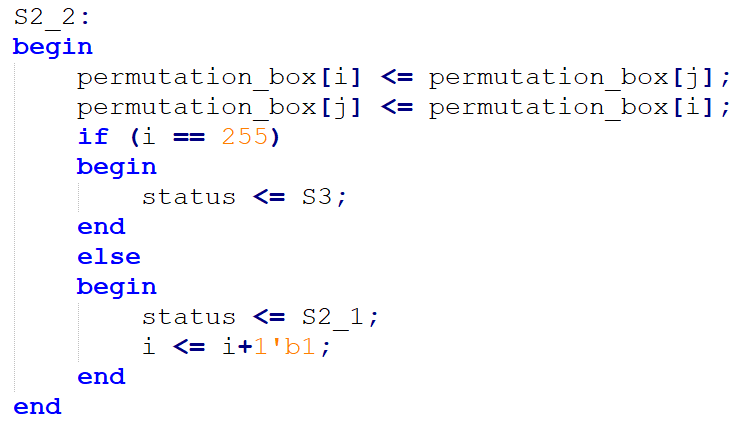


Status S1 is now active. If the key is valid, used\_key is equaled to our key. At the same time, the status is switched. i is set to 0, just like ready\_for\_key and j.

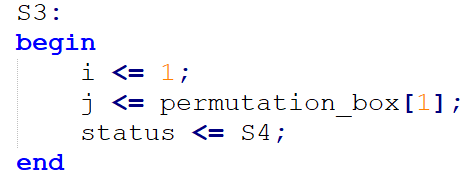
On the other hand, if the key is not valid, ready\_for\_key remains at 1, while the used\_key is reset, as it is not valid. The status is S1 again, to do this step all over again.

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 that are repeated 256 times each one.

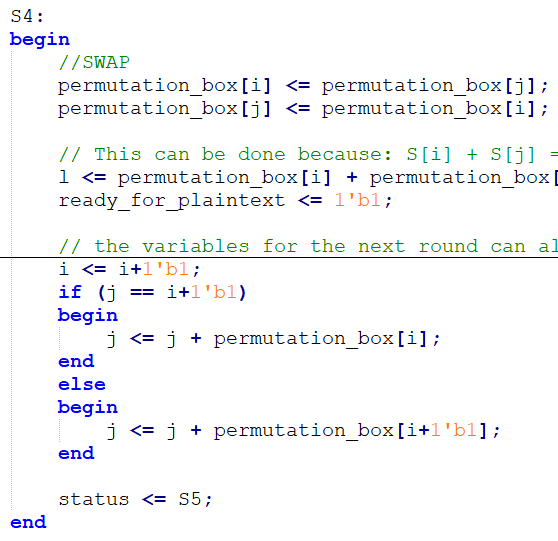
In Step 2\_1 a calculation is conducted to determine the value of j. This operation is equal to the C-Pseudocode 𝑗 = ( 𝑗 + 𝑆[𝑖] + 𝑘𝑒𝑦[𝑖 𝑚𝑜𝑑 16] ) 𝑚𝑜𝑑 256 of the key scheduling algorithm. 



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.

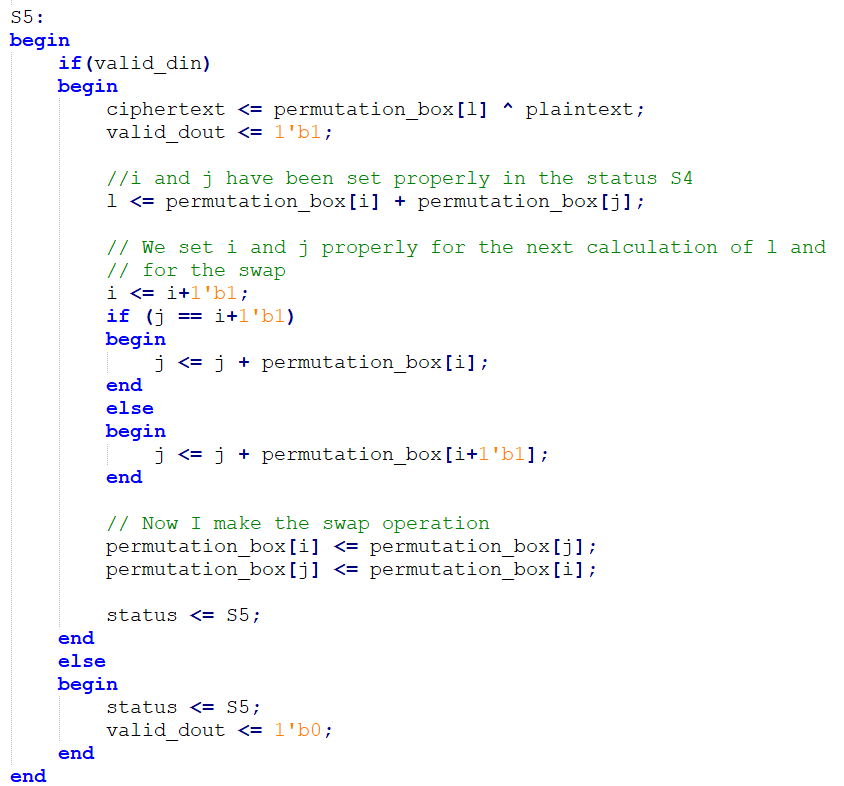


In S3 i is set to 1, while the value of j is set to the value of permuation\_box[1]. This step emulates the first part of the first iteration of the PRGA, updating the reg i and j with the values that they should have before the swap operation.

In Status S4 the swapping of the permuation\_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 l) is then filled with the sum of the permuation\_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.

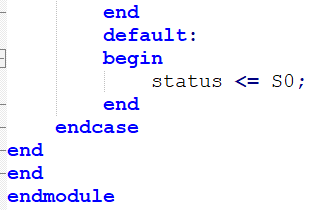


At the status S5 the values provided by the status S4 are the index i and j already set for the swap of the next iteration and the index l of the key used for the encryption of the current iteration.

If we obtain a valid plaintext at input:

* we give at output the ciphertext obtained by the xor of the plaintext and 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 permutation\_box[i] and permutation\_box[j]

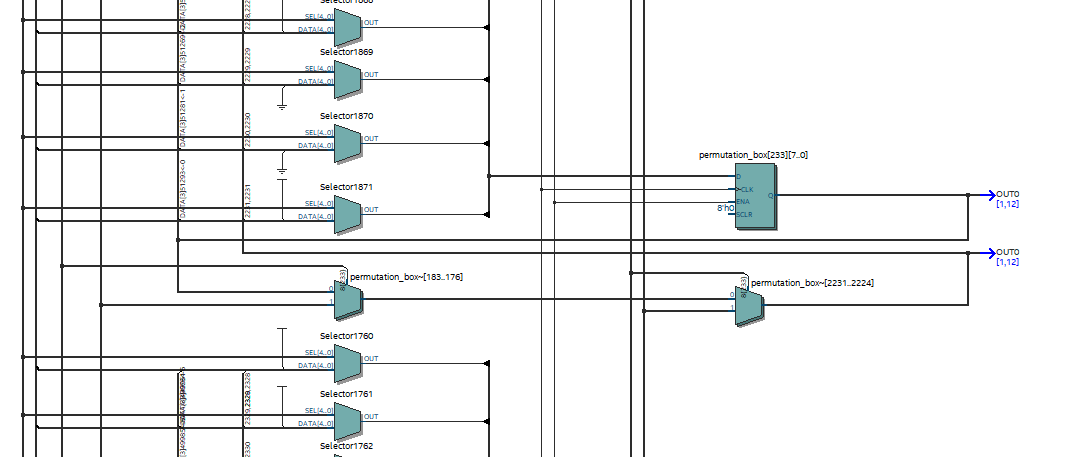
In the case of a non valid input we just reset the reg valid\_out to be sure that the ciphertext is not considered as a good one. Once the module has reached the status S5, it will remain on it until a reset signal or the shutdown of the system occurs, thanks to the fact that all the operations needed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

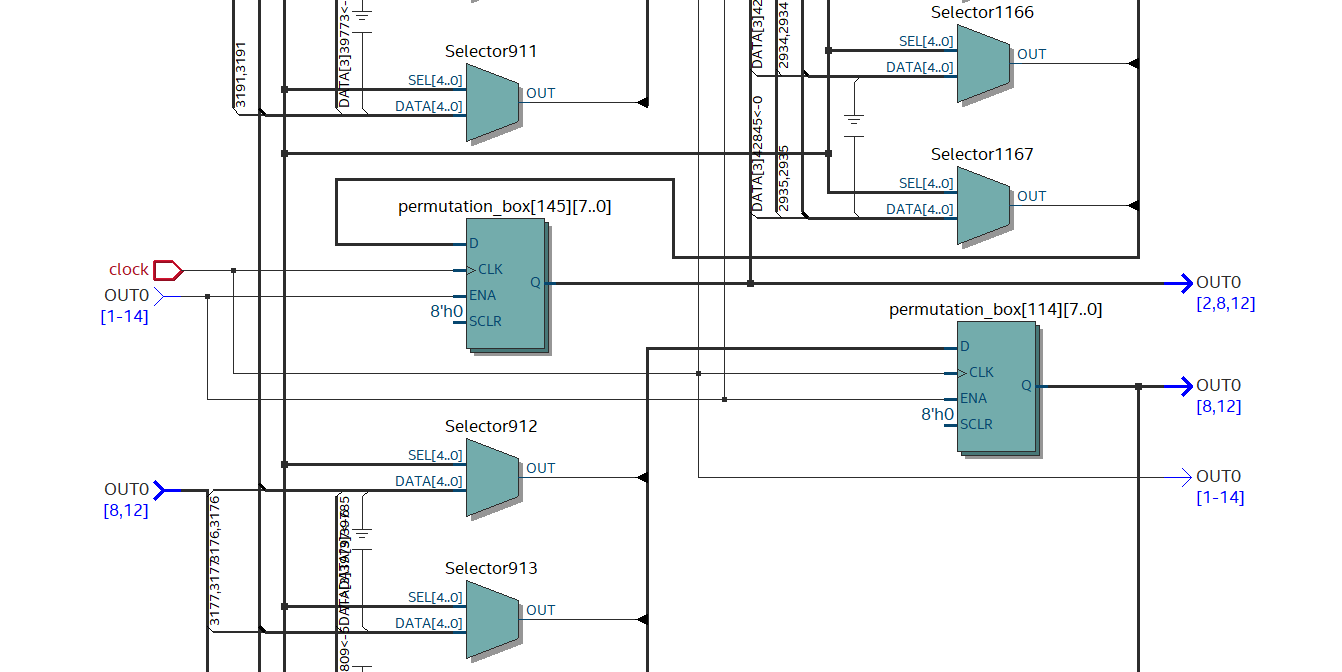


The last noteworthy aspect in this module is the default. In case of any problems, the default is the status S0, e.g. we return to the declaration of the permutation boxes. The process starts over again, from the beginning.

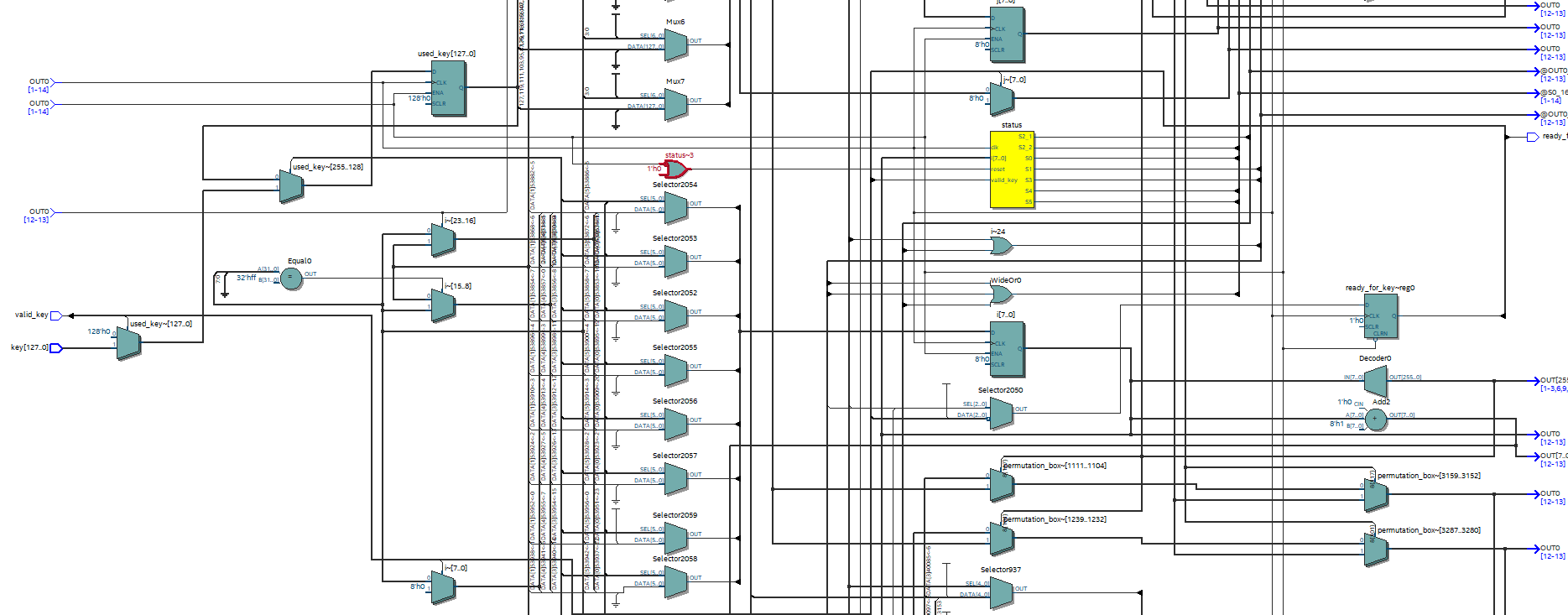
* 1. RTL analysis from Quartus

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 can be seen below, that give an overview of the components defined in the code above.



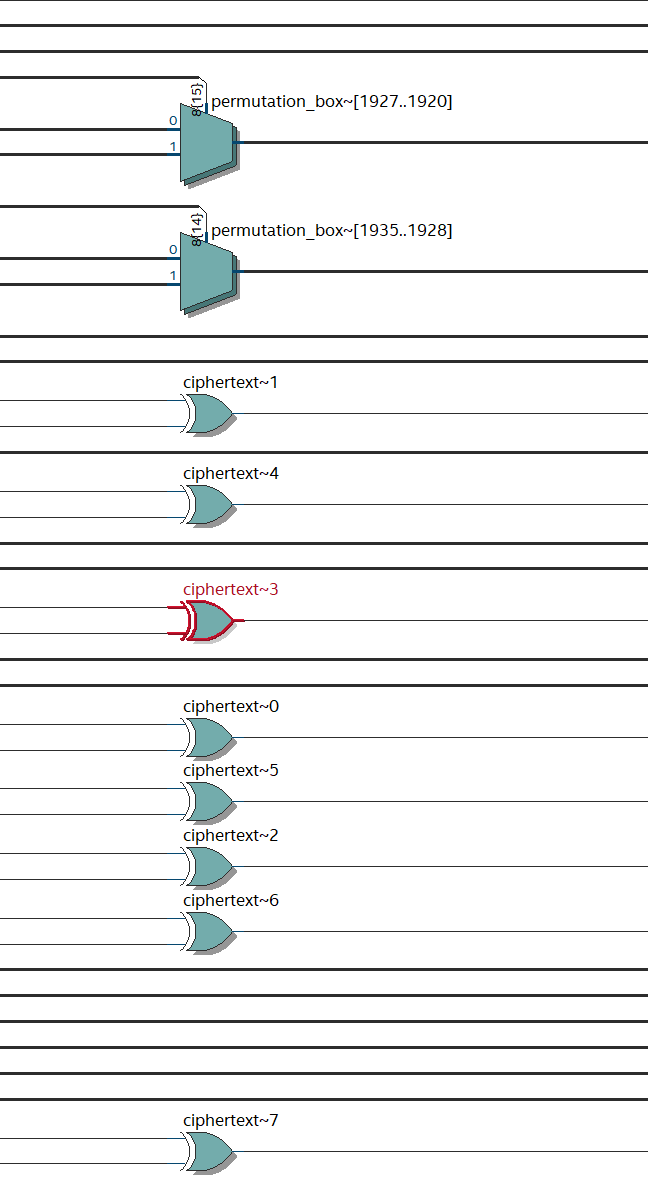


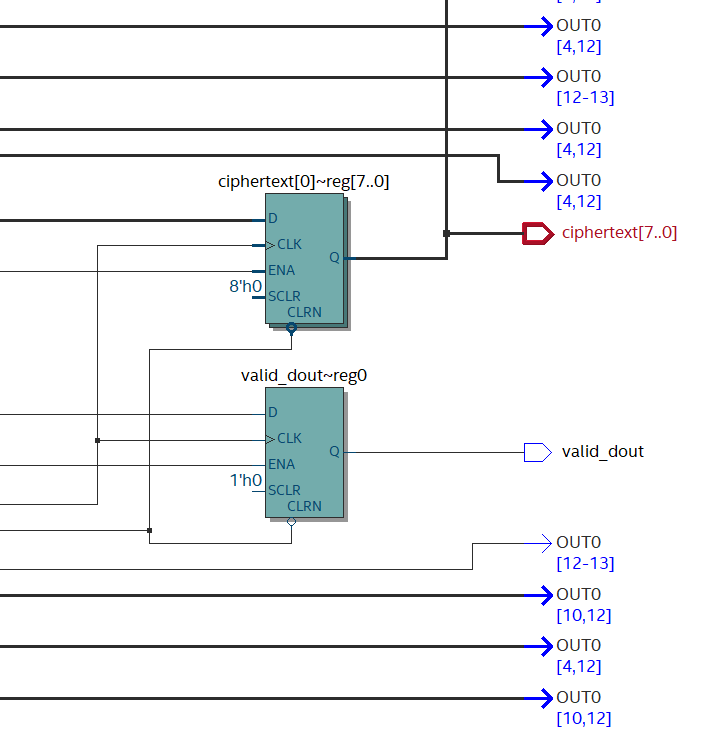
The first screenshot is an exemplary screenshot that shows permutation\_boxes and several selectors. Most of the circuit looks relatively close to this, or the second screenshot just below. The second screenshot also contains the clock input, seen on the left side.



The third screenshot 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.

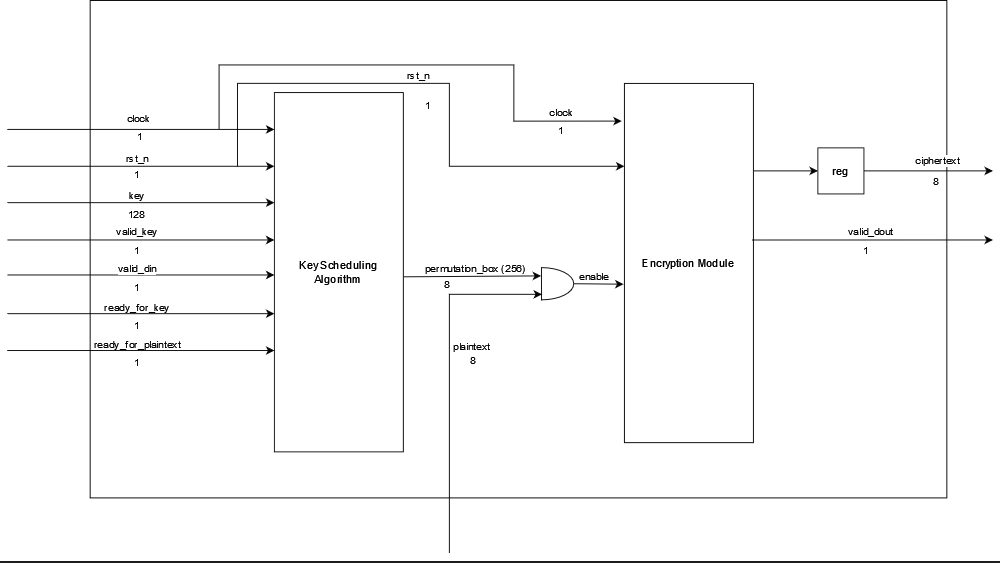
The fourth screenshot gives an overview of 2 permutation boxes, but more importantly the 8 bits of ciphertext that serve as the output. 



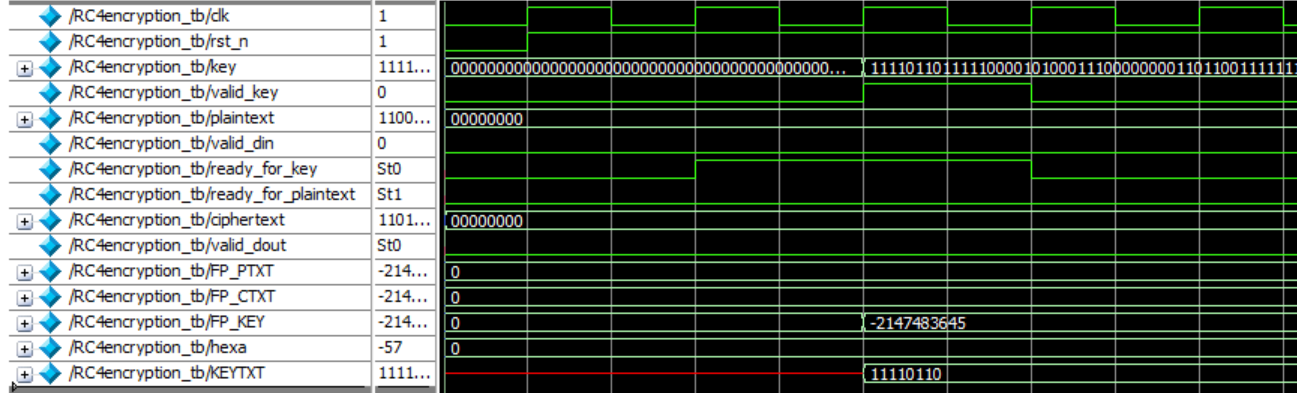
In the last screenshot we see 2 important things. One output is the ciphertext. Its corresponding ciphertext register can also be seen. The valid\_dout register and its output valid\_dout are also visible.

* 1. Schematic

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

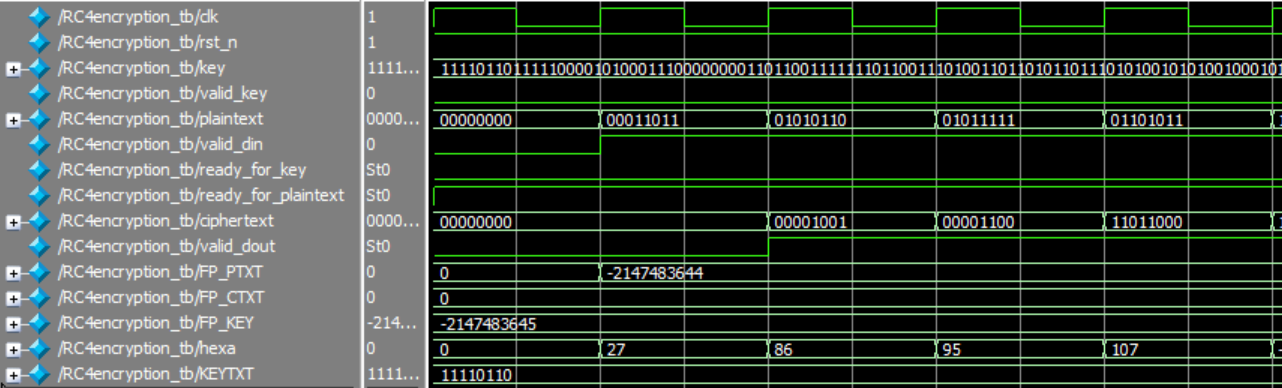


1. **Expected waveforms** 
   1. Modelsim Waveforms
      1. Beginning of process



* **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.
  + 1. Encryption

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

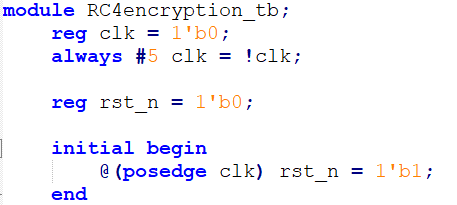


* **clk**: The clock remains the same waveform as above.
* **rst\_n**: Remains the same waveform as above.
* **key**: The key here is still filled with the values that were opened above. In our case, the key is read from a file and its value is cc28bec716a9d4ad4d677f36c051f8f6.
* **valid\_key**: The 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 “**1b**565f6bce1bde2f9e5363c7” (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 “**1b**565f6bce1bde2f9e5363c7”
* **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.
  + 1. Conclusion

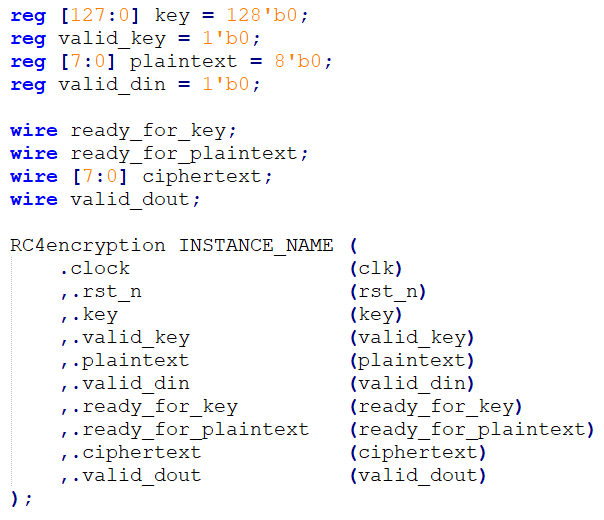
Overall, we can determine that the waveforms are in line with the project requirements.

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.

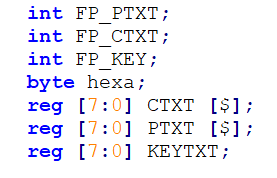
1. **Testbench (block diagram, testbench design choices, comments on test-plan)** 
   1. Architectural and Functional Description of testbench

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. 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.

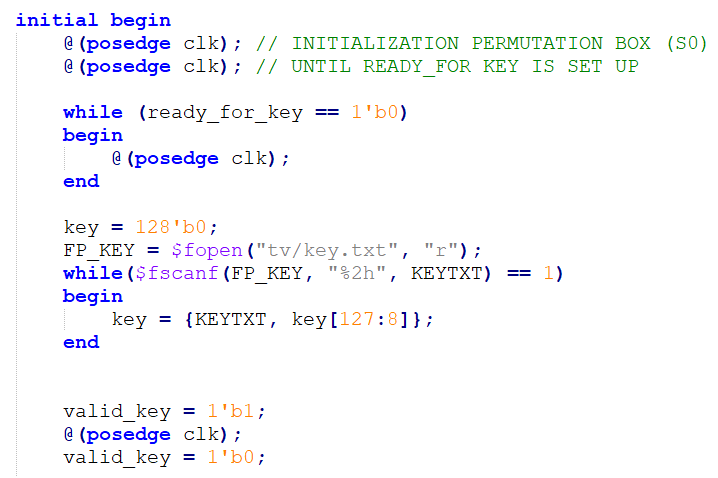
In the next step, several registers and wires are declared. Those include - among others - key, plaintext, valid\_din and valid\_dout. In addition, several instance names are declared that serve as an interface to the main function of the encryption algorithm.



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.

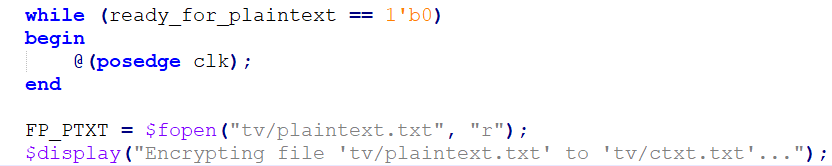


At this point, we enter the main testbench functionality.

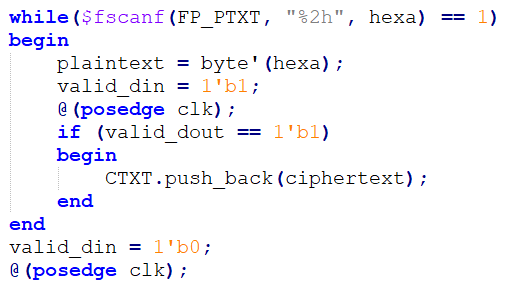


Firstly, we start two clock cycles, to initialize the permutation box, and secondly to wait until the ready\_for\_key register is set up.

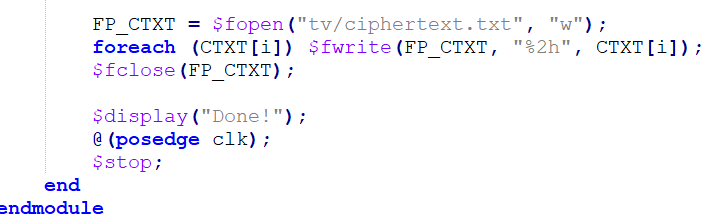
While ready\_for\_key is still at 0, new clock cycles are begun on a constant basis. Once ready\_for\_key is set to 1, we open the text file that contains the key with the “$fopen” command. The key gets read from the file using $fscanf, and the ‘key’ register is properly initialized. At this point, valid\_key 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.

A somewhat similar mechanism that was described for the key above, happens now for the plaintext. Until ready\_for\_plaintext is set to 1, a clock cycle is pushed and the relevant files get opened.

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.



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.

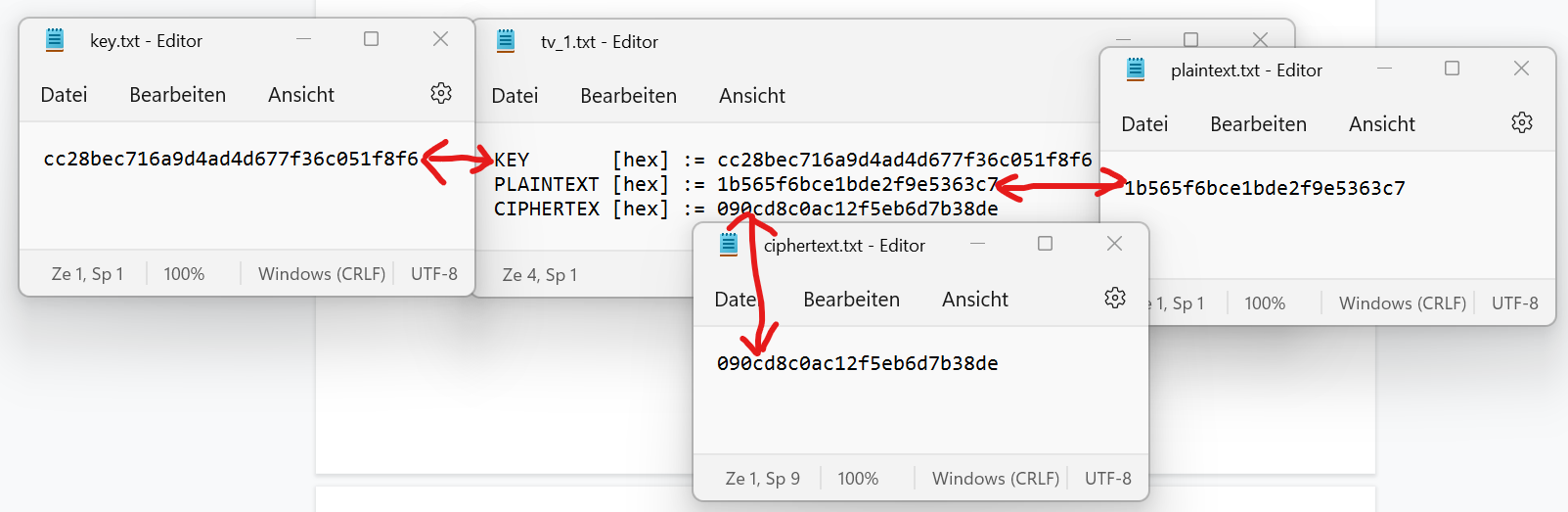


* 1. Verification methodology / approach

Due to simplicity reasons and a lean code, we decided to use a manual approach to verifying the functionality of our circuit. Thanks to Prof. Crocetti, we received several files that we used to assess the correctness.

These 5 files 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.

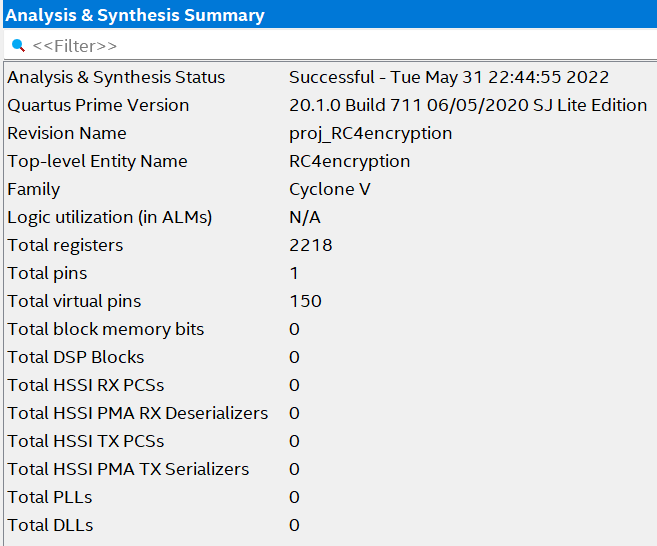


The same process was applied to the 4 other files. All of them worked.

1. **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).

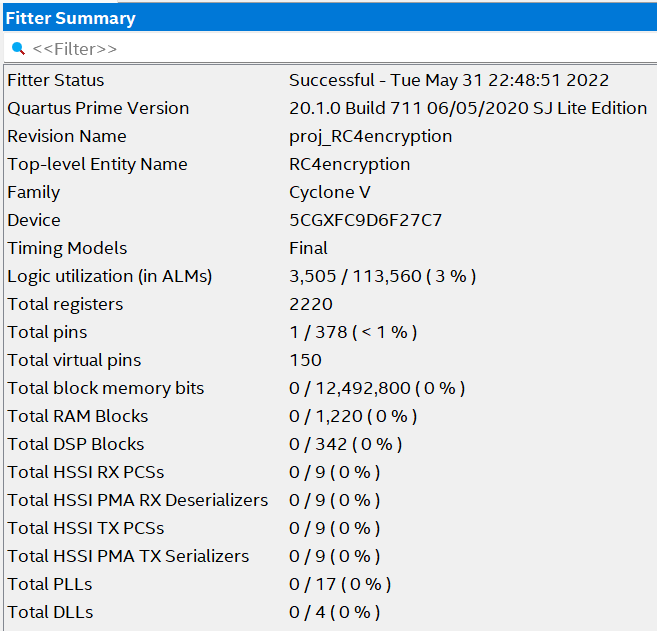
* 1. Analysis & Synthesis Summary



As can be seen in the Analysis & Synthesis summary, 150 pins are assigned. 1 pin is a physical pin, the clock.

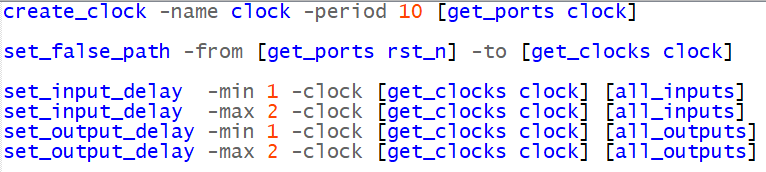
In total we use 2218 registers, this larger number comes due to the nature of the permutation boxes.

* 1. Fitter Summary



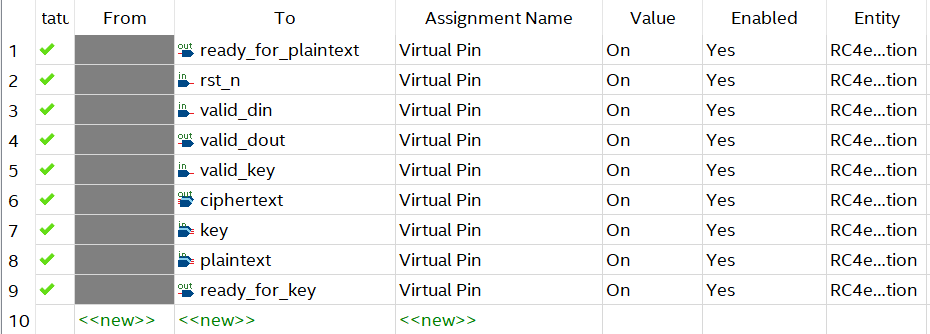
As can be seen in the fitter summary, around 3% of the logic utilization is used. Only 1 of 378 potential pins is used, as the rest of pins are virtual. According to the fitter, there are 2220 total registers.

1. **Static Timing Analysis**
2. SDC file



The SDC file has been kept very simply, but as the following screenshots show, fulfills its purpose perfectly.

1. Virtual Pin Assignment

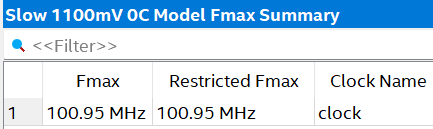


Virtual Pins have been assigned to all components, except the clock. This of course improved the frequency by a considerable margin.

1. Frequency results

The maximum Frequency is 100.95 MHz. In our opinion this is a respectable result. 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.



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

1. Unconstrained Paths Summary

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