

Tiny Encryption Algorithm - Encryption Module

Boschi Gianluca, Carli Francesco, Petri Paola

1 Specification Analysis

1.1 Tiny Encryption Algorithm

The project consists in implementing the encryption module of Tiny Encryption Algorithm (TEA). TEA is a block cipher and operates on a 64-bit data block that will be divided in two 32-bit sub blocks and uses a 128-bit key. It has a Feistel structure with 64 rounds, typically implemented in pairs termed cycles. It has an extremely simple key schedule, mixing all of the key material in exactly the same way for each cycle.

TEA has a few weaknesses: most notably, it suffers from equivalent keys — each key is equivalent to three others, which means that the effective key size is only 126 bits. As a result, TEA is especially bad as a cryptographic hash function.

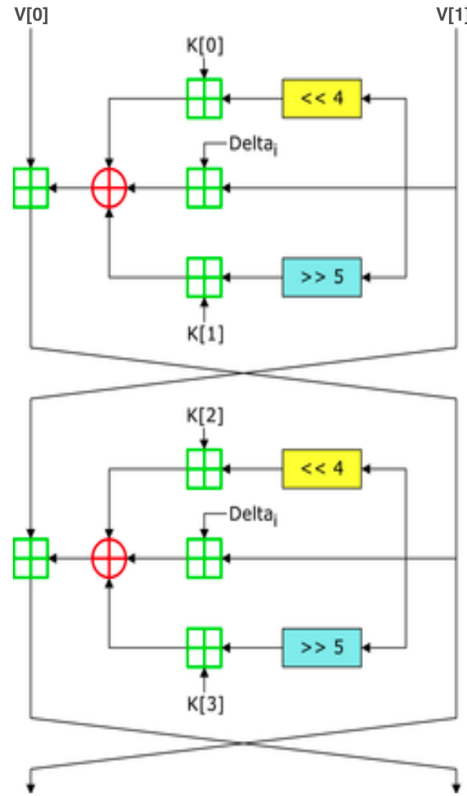


Figure 1: Generic round of TEA

```

void encrypt (uint32_t v[2], const uint32_t k[4]) {
    uint32_t v0=v[0], v1=v[1], sum=0, i;           /* set up */
    uint32_t delta=0x9E3779B9;                       /* a key schedule constant */
    uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];    /* cache key */
    for (i=0; i<32; i++) {                           /* basic cycle start */
        sum += delta;
        v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
    }                                                  /* end cycle */
    v[0]=v0; v[1]=v1;
}

```

Figure 2: C code for encryption function of TEA

The encryption function takes as input a 64-bit block representing the plaintext, that will be split in two 32-bit blocks, and a 128-bit block as a key, that it will be split in four 32-bit blocks. The function uses a constant δ which purpose is to prevent some attacks based on the symmetry of the rounds. The algorithm works on each of 32-bit block in parallel and so the number of rounds is 32 for each block. The operations included in a single round are: xor, right shift, left shift and addition. The algorithm produces as output two 32-bit blocks, that represent the ciphertext.

1.2 Requirements

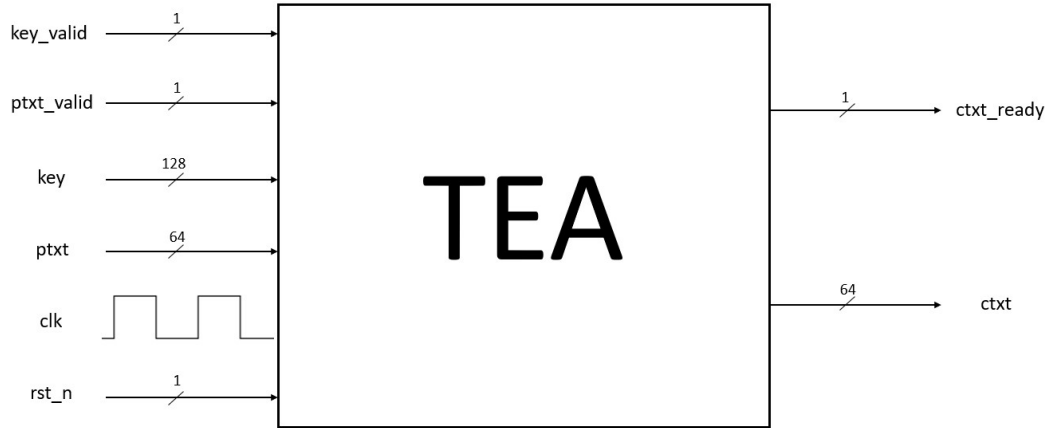


Figure 3: Implementation of TEA encryption module

The encryption module has the following ports:

```

module tiny_encryption_algorithm (
    input          clk          // clock
    ,input          rst_n       // asynchronous reset active low
    ,input          key_valid   // 1 = input data stable and valid, 0 = o.w.
    ,input          ptxt_valid  // 1 = input data stable and valid, 0 = o.w.
    ,input [63:0]   ptxt_blk    // plaintext
    ,input [127:0]  key         // key

    ,output reg [63:0] ctxt_blk  // ciphertext
    ,output reg     ctxt_ready  // 1 = output data stable and valid, 0 o.w.
);

```

- **input clk**: clock signal
- **input rst_n**: asynchronous active-low reset signal. This feature permits to trigger this signal without waiting for the next clock cycle.
- **input key_valid**: signal that has to be asserted when the key is provided in input. If its value is 1, the corresponding input data is valid and stable, otherwise if its value is 0 the key is inconsistent.
- **input ptxt_valid**: signal that has to be asserted when the plaintext is provided in input. If its value is 1, the corresponding input data is valid and stable, otherwise if its value is 0 the plaintext is inconsistent.
- **input [63:0] ptxt_blk**: block of plaintext that will be encrypted by the encryption function of TEA.
- **input [127:0] key**: symmetric key used in the encryption function of TEA.
- **output reg [63:0] ctxt_blk**: block of ciphertext computed by the encryption function of TEA.
- **output reg ctxt_ready**: signal that has to be asserted when the ciphertext is available on the corresponding output port. If its value is 1, the output data is valid and stable, otherwise if its value is 0 the ciphertext is inconsistent.

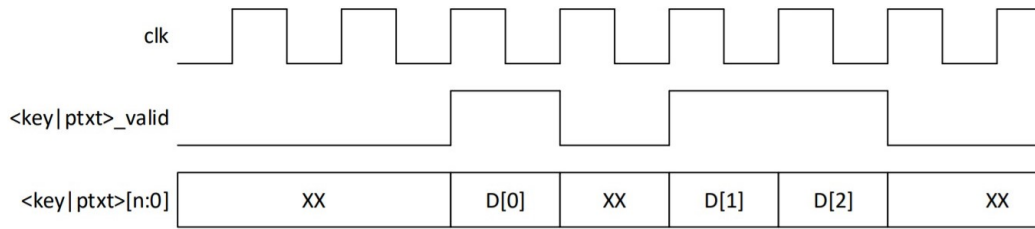


Figure 4: Example of waveform expected considering key_valid and ptxt_valid signals

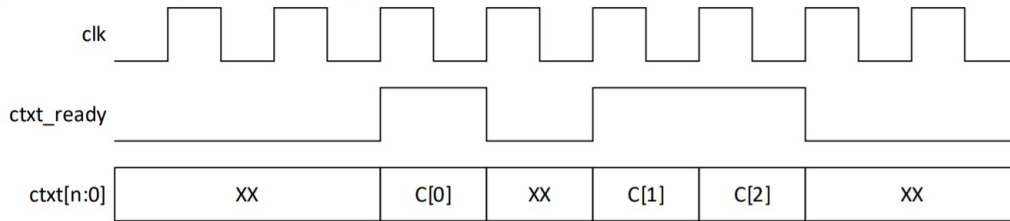


Figure 5: Example of waveform expected considering ctxt_ready signal

2 Block diagram and design choices (RTL design module)

First of all, before to start the implementation of the module, several high-level schema have been drawn to show in more details how the entire hardware module should have been.

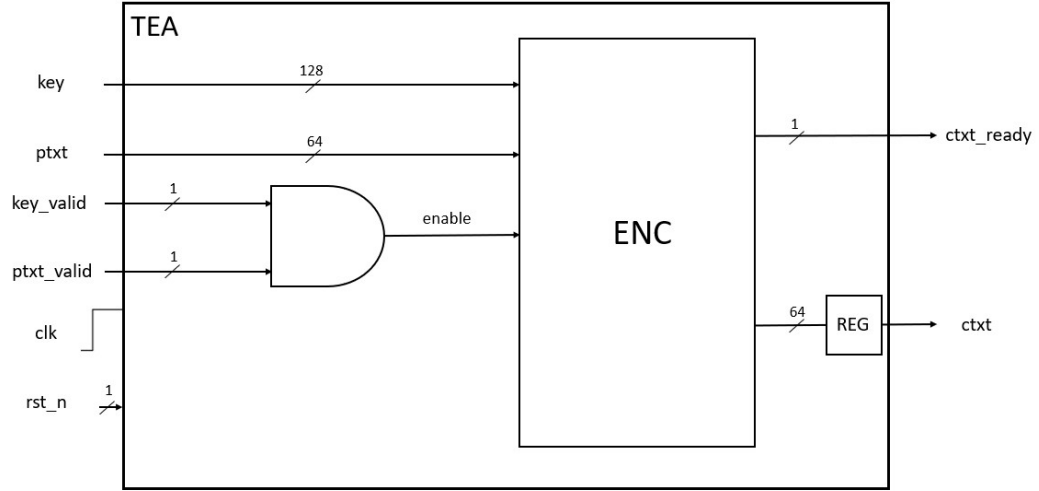


Figure 6: High-level schema of TEA

In the previous schema are shown both input and output ports. The input signals *key* and *ptxt* are used by the encryption module, only if both *key_valid* and *ptxt_valid* have a logic value to 1. For this reason, in the previous schema it has decided to represent this behavior using an *AND* port and its output is used like a sort of enable signal to trigger the encryption. The encryption result is stored inside a 64-bit register whose data will be valid and available when *ctxt_ready* will have a logic value to 1.

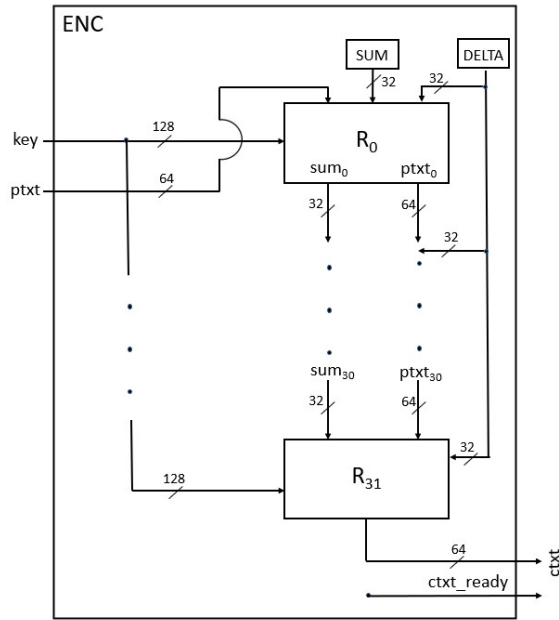


Figure 7: High-level schema of encryption module of TEA

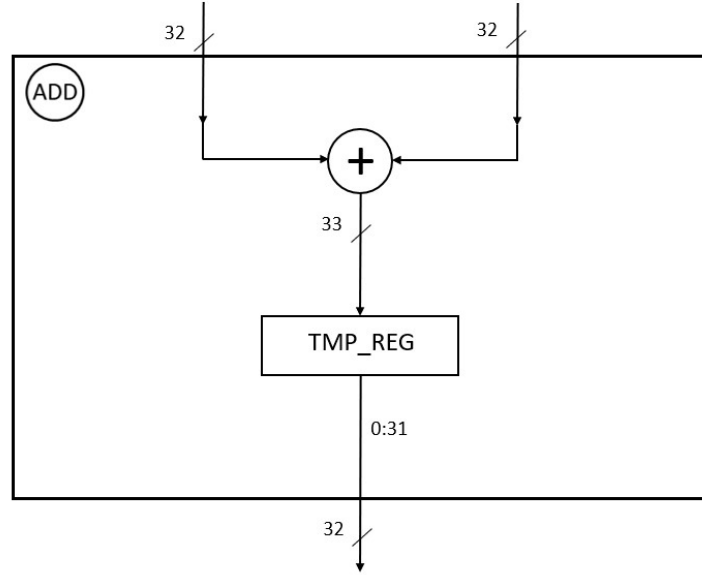


Figure 9: Schema of an addition between 32-bit operands

The custom addition takes as input two 32-bit operands and stores the result in a 33-bit temporary register. The extra bit is needed to maintain the carry. The final result depends from the least significant 32-bit taken from the 33-bit temporary register.

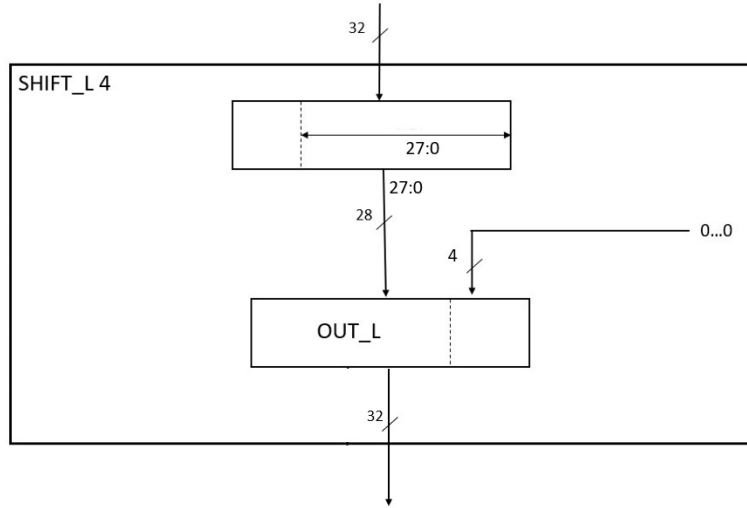


Figure 10: Schema of 4-bit left shift on a 32-bit operand

The custom 4-bit left shift takes as input a 32-bit value that it is stored in a 32-bit temporary register. From this register the least significant 28-bit are taken and then it is used the concatenation operator to insert 4-bit at zero. This new value is the output of the 4-bit left shift.

The custom 5-bit right shift in Figure 11 has been implemented following the same concepts.

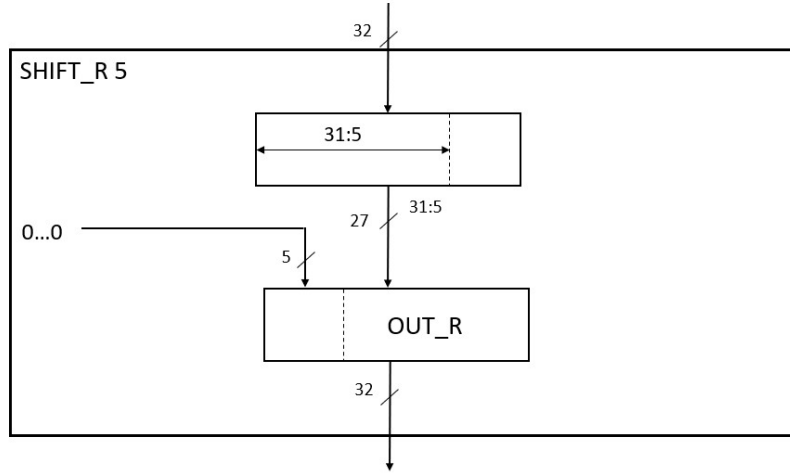


Figure 11: Schema of 5-bit right shift on a 32-bit operand

3 Expected waveforms

In this section different waveforms, obtained from two different testbenches (Section 4), are included in order to show the behavior of the encryption module of TEA.

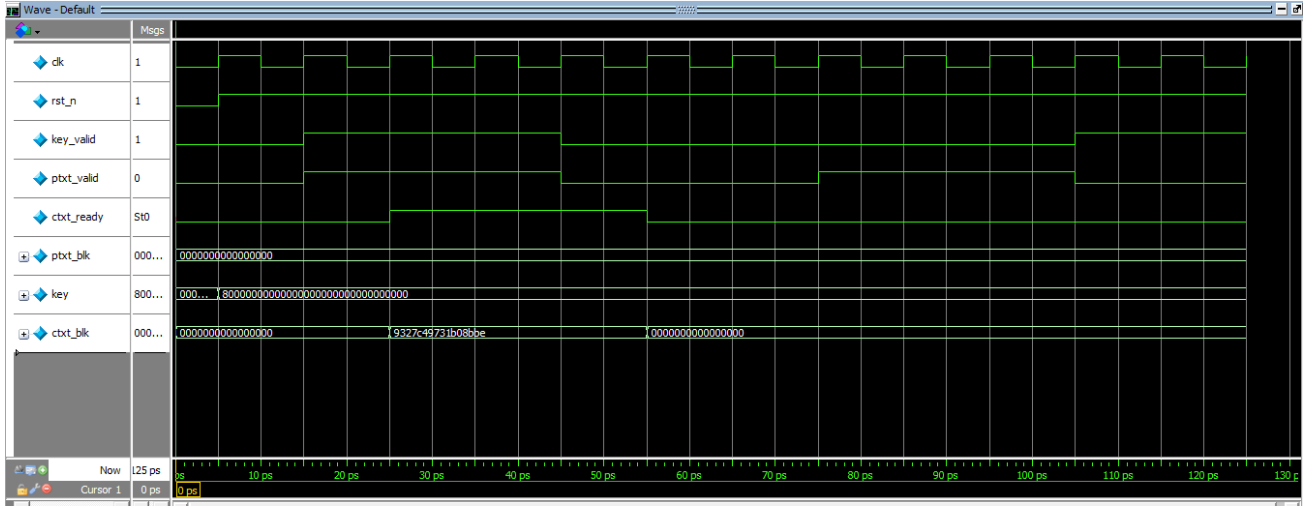


Figure 12: A waveform obtained from the corner cases' testbench.

Observing Figure 12, at the beginning of the simulation, the *rst_n* signal switches from logic 0 to logic 1 and sets the value of *ctxt_blk* and *ctxt_ready* to a default value. It can be done a comparison between the signals *ptxt_valid*, *key_valid*, *ptxt* and *key* and it is possible to see the corresponding behavior of the signal *ctxt_ready* from the previous ones. More in details, *ctxt_ready* has a value to logic 1 only when both *ptxt_valid* and *key_valid* have a value to logic 1. Furthermore, it can be observed that the signal *ctxt*, the final result of the encryption process, is valid and available only when *ctxt_ready* has a logic value to 1.

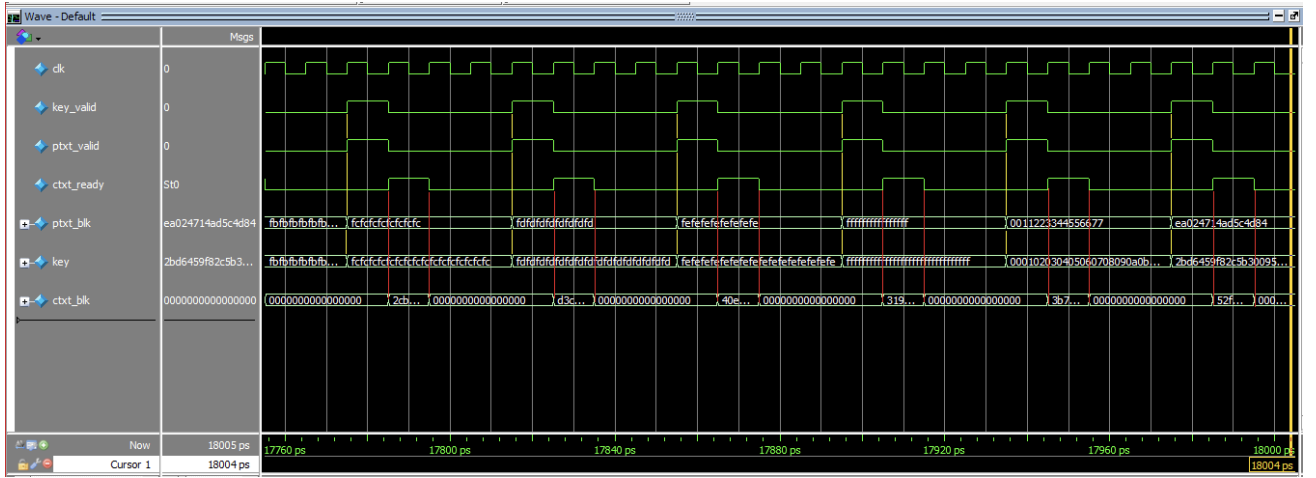


Figure 13: A waveform obtained from the testbench that checks the encryption

The waveform in Figure 13 shows how the encryption module of TEA works, taking as inputs different values of plaintext and key from a specific file. In particular, is it possible to see that the value of *ptxt* and *key* changes respectively when the corresponding flag switches from a logic 0 to a logic 1, making them valid and available. This behaviour can be observed looking at the yellow lines on the positive edge of *ptxt_valid* and *key_valid*. Another important fact to observe, looking at the red lines, is that the computed ciphertext, represented by the signal *ctxt*, is valid and available only when the corresponding flag *ctxt_ready* has a value to logic 1.

4 Testbench

It has been developed two main testbenches to test and check the right functionality of the encryption module of TEA: one test checks the corner cases regards *ptxt_valid* and *key_valid* signals, the other one tests the right behavior of the module taking a provided test vector as input.

The test vector is composed by three different files: one contains a set of plaintexts, another one contains a set of keys and the last one contains a set of expected ciphertexts.

4.1 Testbench: corner case

This testbench is used to check the right behavior of the module related to all the possible values that *ptxt_valid* and *key_valid* can assume. Those two inputs are 1-bit signals, so in the testbench there are four different cases. The right concept on which this testbench has been developed is that *ctxt_ready* is at logic value 1 only if both *ptxt_valid* and *key_valid* were previously at logic value 1.

In this testbench it has been used a single plaintext and a single key from the test vector, because its behavior doesn't depend from the value of a plaintext or of a key. To analyze this concept it has been developed another testbench that will be analyzed in the following subsection.

```
# PTXT_VALID: 1 - KEY_VALID: 1 - CTXT_READY: 1 - OK
# PTXT_VALID: 0 - KEY_VALID: 0 - CTXT_READY: 0 - OK
# PTXT_VALID: 1 - KEY_VALID: 0 - CTXT_READY: 0 - OK
# PTXT_VALID: 0 - KEY_VALID: 1 - CTXT_READY: 0 - OK
```

Figure 14: Output of testbench from ModelSim

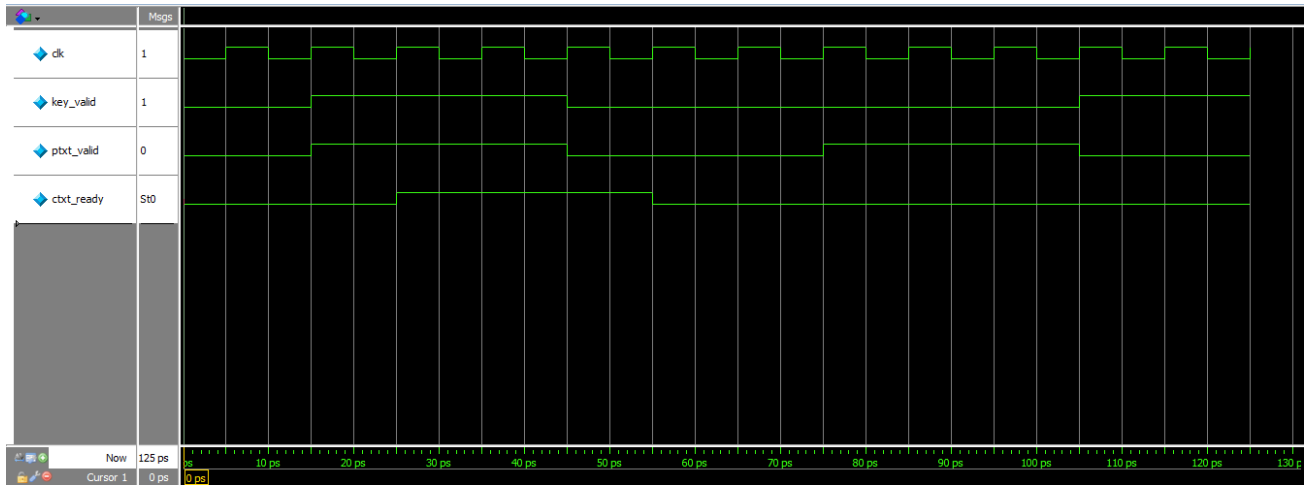


Figure 15: Waveform that shows the right behavior of signal *ctxt_ready*

From Figure 15 it is possible to see that there is only one case where *ctxt_ready* is at logic value 1 and that is because only in that moment both *ptxt_valid* and *key_valid* are at logic value 1.

4.1.1 Implementation of test

```
// PTXT_VALID = 1 - KEY_VALID = 1
@(posedge clk);
ptxt_valid = 1'b1;
key_valid  = 1'b1;

@(posedge clk);
@(posedge clk);
$display("PTXT_VALID: %b - KEY_VALID: %b - CTXT_READY: %b - %5s",
        ptxt_valid, key_valid, ctxt_ready, ctxt_ready == 1'b1 ? "OK" : "ERROR");
```

Above it is reported a portion of source code used in the testbench. At the beginning it is used `@(posedge clk)` to align with the current clock cycle, then *ptxt_valid* and *key_valid* are initialized. To check the value of *ctxt_ready* it is necessary to wait for two clock cycles: in the first one it is updated the value of a temporary internal register, while in the second one it is updated the value of the output signal *ctxt_ready*.

4.2 Testbench: encryption

This testbench has been developed to test the right functionality of the encryption module of TEA. Its main purpose is to check if, given a certain plaintext, an encryption key and an expected ciphertext, the computed result will be equal as the expected one. This testbench takes its inputs from three different files of test vectors, respectively one for the plaintext, one for the key and one for the expected ciphertext.

```
#      PLAINTEXT |      CIPHERTEXT | EXPECTED CIPHERTEXT
# 0000000000000000 9327c49731b08bbe 9327c49731b08bbe OK
# 0000000000000000 77316e05c9ed06fb 77316e05c9ed06fb OK
# 0000000000000000 d6b88f1b6b1e5f3f d6b88f1b6b1e5f3f OK
# 0000000000000000 039ff567e117b685 039ff567e117b685 OK
# 0000000000000000 56b44dc7d3cbe7cf 56b44dc7d3cbe7cf OK
# 0000000000000000 7230929cc60bd350 7230929cc60bd350 OK
# 0000000000000000 08e7c8795485edf0 08e7c8795485edf0 OK
# 0000000000000000 7cc227951deb2351 7cc227951deb2351 OK
# 0000000000000000 691d4a5c210cd865 691d4a5c210cd865 OK
# 0000000000000000 7421dbb014a8da4e 7421dbb014a8da4e OK
# 0000000000000000 3446cd86de5ee499 3446cd86de5ee499 OK
# 0000000000000000 fa79b09fb5a395f6 fa79b09fb5a395f6 OK
# 0000000000000000 e86f97ed5579c450 e86f97ed5579c450 OK
# 0000000000000000 7b27e59efe093a9d 7b27e59efe093a9d OK
# 0000000000000000 5e34d77e40c1c08e 5e34d77e40c1c08e OK
# 0000000000000000 7f904680db138017 7f904680db138017 OK
# 0000000000000000 67a6286f3e956426 67a6286f3e956426 OK
# 0000000000000000 731134baddflaf56 731134baddflaf56 OK
# 0000000000000000 2aece694cc553274 2aece694cc553274 OK
# 0000000000000000 971cbb6f10a84aab 971cbb6f10a84aab OK
# 0000000000000000 81f67368204d961d 81f67368204d961d OK
```

Figure 16: Output of testbench from ModelSim

In Figure 16 it is shown a partial output of the testbench, from whose it's possible to see how the computed ciphertexts (second column) are equal to the expected ciphertexts (third column). Basing on the provided test vector, 449 different tests have been ran and all of them have reached a positive result.

For more details about what Figure 17 shows, please see Section 3.

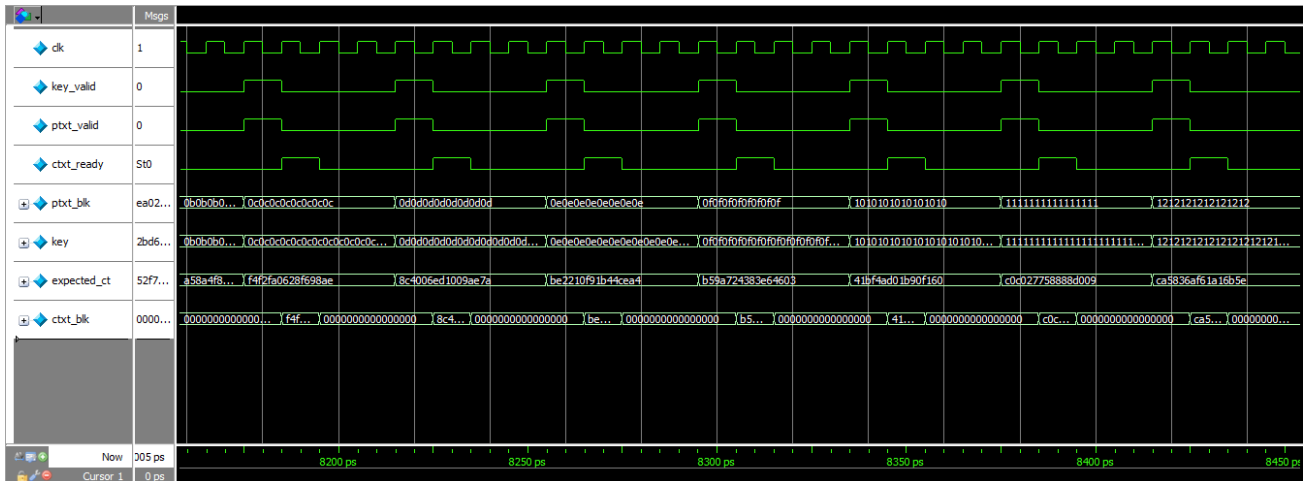


Figure 17: Waveform that shows the right behavior of the encryption module

4.2.1 Implementation of test

```
always @(posedge clk) begin
    scan_file = $fscanf(key_file, "%h\n", tmp_key);
    scan_file = $fscanf(pt_file, "%h\n", tmp_pt);
    scan_file = $fscanf(ct_file, "%h\n", tmp_ct);
    if (!$feof(key_file) && !$feof(pt_file) && !$feof(ct_file)) begin
        @(posedge clk);
        ptxt_blk    = tmp_pt;
        key         = tmp_key;
        ptxt_valid   = 1'b1;
        key_valid    = 1'b1;

        expected_ct = tmp_ct;

        @(posedge clk);
        wait (ctxt_ready === 1'b1);
        ptxt_valid = 1'b0;
        key_valid  = 1'b0;

        @(posedge clk);
        $display("%h %h %h %-5s", ptxt_blk, ctxt_blk, expected_ct,
            expected_ct === ctxt_blk ? "OK" : "ERROR");
    end
    else begin
        $display("End of files");
        $stop;
    end
end
end
```

From the always block above it is possible to see how this testbench works. At the beginning, it reads from the files of the test vector a plaintext, a key and an expected ciphertext. Then, if EOF for those files is not reached yet, the plaintext and the key are used to initialize the input of the encryption module, and the respectively validity signals are set to logic value 1. After that, it is necessary to wait until *ctxt_ready* will switch from 0 to 1 because at that moment the computed ciphertext will be available and valid. Before to compare the processed ciphertext and the expected one, the two validity signals are set to logic value 0 to perform the next encryption. At the end, the computed ciphertext is compared with the expected ciphertext.

5 Implementation of RTL design on FPGA and results

This section contains the steps of Analysis & Synthesis and Fitter (Place & Route) that have been performed using Quartus. It is possible to see, from the following reports, that the chosen FPGA is from the Cyclone V family, in particular the model is 5CGXFC9D6F27C7. Devices from this family have between their characteristics low costs and low power consumption.

Analysis & Synthesis Summary	
<<Filter>>	
Analysis & Synthesis Status	Successful - Mon Jul 26 14:42:42 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	tiny_encryption_algorithm
Top-level Entity Name	tiny_encryption_algorithm
Family	Cyclone V
Logic utilization (in ALMs)	N/A
Total registers	65
Total pins	1
Total virtual pins	260
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 18: Analysis & Synthesis summary from Quartus

From the Fitter summary in Figure 19 it is possible to see that the amount of logic resources used by the implemented module are less than 5%. This result outlines that this type of FPGA isn't the best choice on which implement the encryption module of TEA, because there would be a certain waste of logical resources.

Fitter Summary	
<<Filter>>	
Fitter Status	Successful - Mon Jul 26 14:54:16 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	tiny_encryption_algorithm
Top-level Entity Name	tiny_encryption_algorithm
Family	Cyclone V
Device	5CGXFC9D6F27C7
Timing Models	Final
Logic utilization (in ALMs)	5,124 / 113,560 (5 %)
Total registers	65
Total pins	1 / 378 (< 1 %)
Total virtual pins	260
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 19: Fitter summary from Quartus

The number of total pins is 1 and it is due to the clock pin. The other input and output pins are virtual, because they could be triggered by other logical resources, so they are not considered in the total amount of pins. Anyway, in a real implementation of this module, these pins will be physical and about 70% of them will be used.

6 Static Timing Analysis (STA)

Static Timing Analysis is a very important step of the design flow because it allows to check if the implemented module behaves in the correct way, respecting some additional requirements and constraints.

To provide some time constraints about input and output signals of the implemented module, it has been written a `.sdc` file. It is necessary to re-run the entire process of Analysis & Synthesis and Fitter in order to allow the synthesis engine to check the respect of the constraints.

6.1 Synopsys Design Constraints File

```
create_clock -name clk -period 10 [get_ports clk]
set_false_path -from [get_ports rst_n] -to [get_clocks clk]
set_input_delay -min 1 -clock [get_clocks clk] [get_ports {rst_n key_valid ptxt_valid ptxt_blk[*] key[*]}]
set_input_delay -max 2 -clock [get_clocks clk] [get_ports {rst_n key_valid ptxt_valid ptxt_blk[*] key[*]}]
set_output_delay -min 1 -clock [get_clocks clk] [get_ports {ctxt_ready ctxt_blk[*]}]
set_output_delay -max 2 -clock [get_clocks clk] [get_ports {ctxt_ready ctxt_blk[*]}]
```

Figure 20: Containt of `.sdc` file

In the Figure above it is shown the `.sdc` file developed to create several time constraints. The first line has the function to create a clock object with a period of 10 ns and link it to the input port of the module that concerns the clock signal.

The second line says that the synthesis engine doesn't have to check the respect of the time constraints on the reset signal: this signal is asynchronous and it is independent from the clock signal during its activation.

In the rest of the `.sdc` file there are two more constraints that concern input and output ports of the module. It is important to consider also these constraints because input and output ports of a module could have delays. In particular, both input and output ports have a minimum and a maximum delay: a minimum delay has been set to 10% of the clock period, while a maximum delay has been set to the 20% of the clock period.

To make sure that the `.sdc` file has been developed and added in the right manner to the project of the module on Quartus, a user can check if some unconstrained paths are still present inside the Static Timing Analysis summary: if this number is zero, the file of time constraints has been added properly.

6.2 Virtual pins

	tatu	From	To	Assignment Name	Value	Enabled	Entity
1	✓		out ctxt_ready	Virtual Pin	On	Yes	tiny_en...gorithm
2	✓		in key_valid	Virtual Pin	On	Yes	tiny_en...gorithm
3	✓		in ptxt_valid	Virtual Pin	On	Yes	tiny_en...gorithm
4	✓		in rst_n	Virtual Pin	On	Yes	tiny_en...gorithm
5	✓		out ctxt_blk	Virtual Pin	On	Yes	tiny_en...gorithm
6	✓		in key	Virtual Pin	On	Yes	tiny_en...gorithm
7	✓		in ptxt_blk	Virtual Pin	On	Yes	tiny_en...gorithm
8		<<new>>	<<new>>	<<new>>			

Figure 21: Report of virtual pins of the module

In Figure 21 input and output ports of the module have been specified as virtual. That's due to the fact that the right physical location of every pin is unknown because this module could be driven by other logic resources on the FPGA. It is possible to see from the previous report that the clock signal hasn't be specified as virtual because it is a physical signal coming from outside the module.

6.3 Analysis of frequencies


Slow 1100mV 85C Model Fmax Summary				
 <<Filter>>				
	Fmax	Restricted Fmax	Clock Name	Note
1	3.06 MHz	3.06 MHz	clk	

Figure 22: Slow 1.1V 85C model max frequency


Slow 1100mV 0C Model Fmax Summary				
 <<Filter>>				
	Fmax	Restricted Fmax	Clock Name	Note
1	3.01 MHz	3.01 MHz	clk	

Figure 23: Slow 1.1V 0C model max frequency

From the previous summaries obtained from the Static Timing Analysis process on Quartus, it is possible to observe that the module supports a maximum frequency of 3.06 MHz considering a model that works at 85°C and a maximum frequency of 3.01 MHz considering a model that works at 0°C.

Despite there is a frequency greater than the other, it has to be considered the lower one of 3.01 MHz because this is the frequency at which the module is able to work in the worst case.

In general, the maximum frequency isn't very high and it is due to how the module has been implemented and developed (for more details see RTL Viewer tool on Quartus). One possible improvement, that increase also the latency, could be to design a single sub-module that implements only one encryption round and use it 32 times with a counter.