

Tiny Encryption Algorithm - Encryption Module

Boschi Gianluca, Carli Francesco, Petri Paola

University of Pisa
M.Sc. Cybersecurity

Hardware and Embedded Security course

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 are divided in two 32-bit sub blocks and uses a 128-bit key. It has a 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.

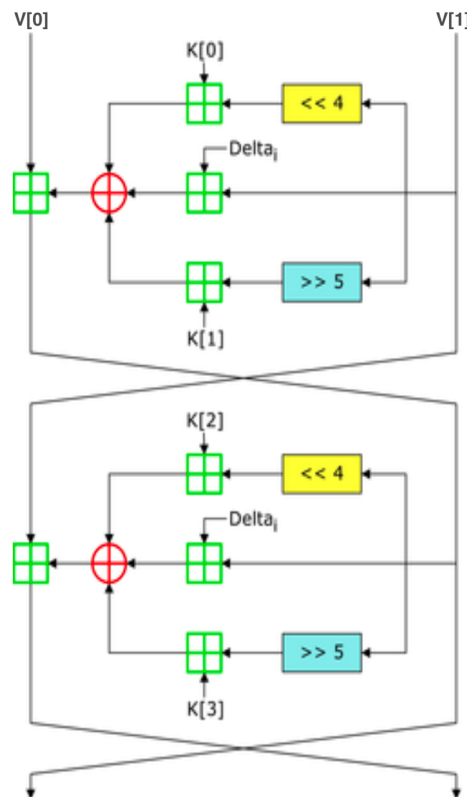


Figure 1: Generic round of TEA

```

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 is split in two 32-bit blocks ($V[i]$ and $v[i]$, for $i = 0,1$, respectively in Figure 1 and Figure 2), and a 128-bit block as a key, that it is split in four 32-bit blocks ($K[i]$ and $k[i]$, for $i = 0,1,2,3$, respectively in Figure 1 and Figure 2). The function uses a constant called Δ_i or δ_i , respectively in Figure 1 and Figure 2, 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 (red circle in Figure 1), right shift (light blue box in Figure 1), left shift (yellow box in Figure 1) and addition (green box in Figure 1). 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. The asynchronous property of this port 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 to 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.

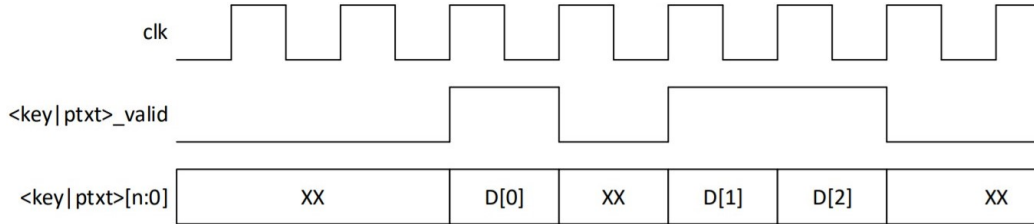


Figure 4: Example of waveform expected considering key_valid and ptxt_valid signals

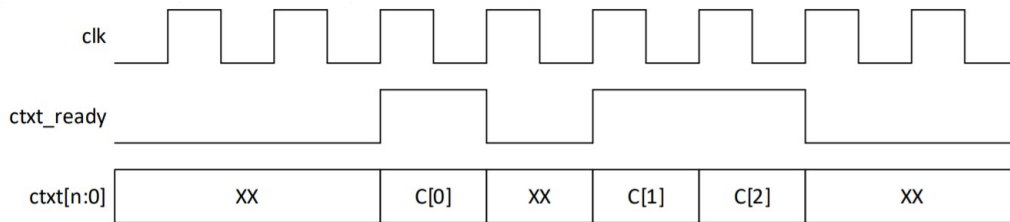


Figure 5: Example of waveform expected considering ctxt_ready signal

2 Block diagram and design choices (RTL design module)

In this section, it is shown the schematic of the implemented TEA module following a top-down approach, considering both the schematic of the top-level entity and schematics of sub-blocks.

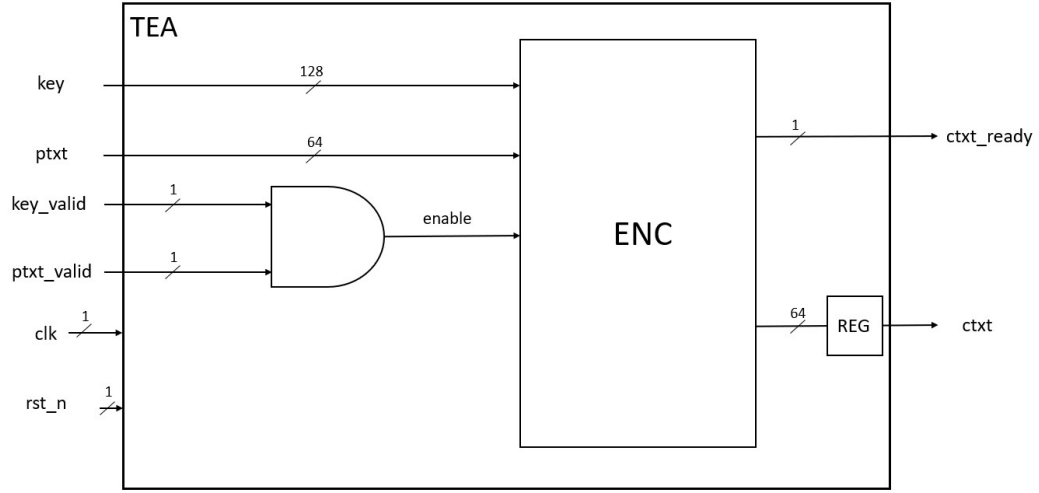


Figure 6: High-level schematic of TEA

Figure 6 shows 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. Thus such signals (i.e. *key_valid* and *ptxt_valid*) are mixed through an AND gate, whose output is used as enable signal to trigger sub-module performing the encryption function. The encryption result is stored into a 64-bit register whose data is valid and available when *ctxt_ready* has a logic value to 1.

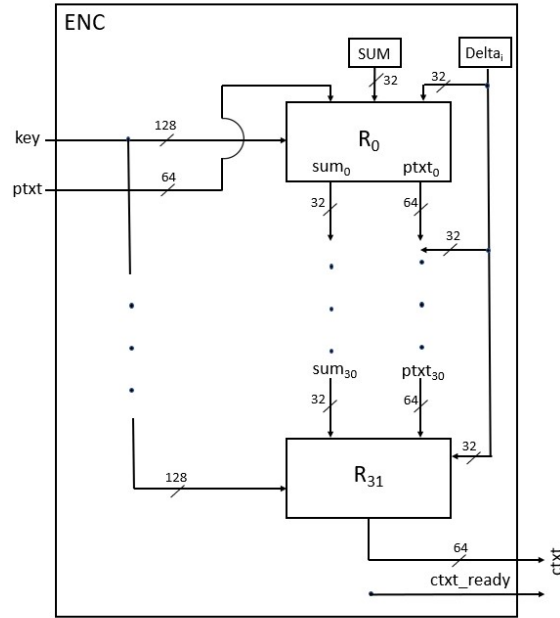


Figure 7: High-level schematic of encryption module of TEA

Figure 7 illustrates the high-level schematic of the encryption module of TEA. It is composed by 32 rounds, from R_0 up to R_{31} , and each round takes as input the 128-bit key and the constant Δ_i . Moreover, round from 0 to 30 have two outputs, a 32-bit sum and a 64-bit modified plaintext, being the outputs of round i is used as inputs in the round $i+1$. The last encryption round (R_{31}) has a single output that is the computed ciphertext. The encryption module has two outputs: a 64-bit block of ciphertext and a flag *ctxt_ready* that is assigned at logic value to 1 after the last round.

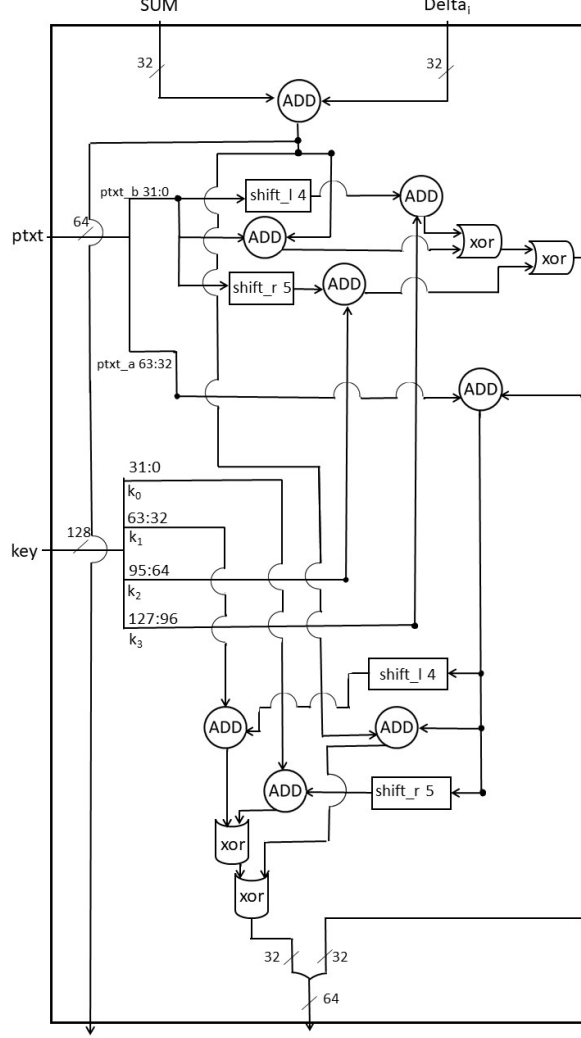


Figure 8: High-level schematic of a single encryption round of TEA

Figure 8 shows a single round of encryption. The 64-bit block of plaintext is divided into two 32-bit blocks, called respectively *ptxt.a* and *ptxt.b*, and the 128-bit key is divided into four 32-bit subkeys. The value of *sum* is taken as input from the previous round and it is added to the constant Δ_i . The encryption process starts on *ptxt.b*: it is left-shifted by 4 bits and right-shifted by 5 bits and the results are used in two different addition with two subkeys, called k_0 and k_1 . *ptxt.b* is also added to *sum* and all the three intermediate results are combined with a xor operation. The 32-bit result of the xor is an output of the encryption round but it is also used in the encryption process on *ptxt.a*. The value of this register is added with the previous result of the xor and then the new value is subjected to other shifts, additions (that use subkeys k_2 and k_3) and xor. The 32-bit result of these operations is another part of the final output of an encryption round.

Each round contains custom operations for additions, left shifts and right shifts. To avoid undefined behaviors, such operations are implemented as illustrated, respectively in Figure 9, Figure 10 and Figure 11.

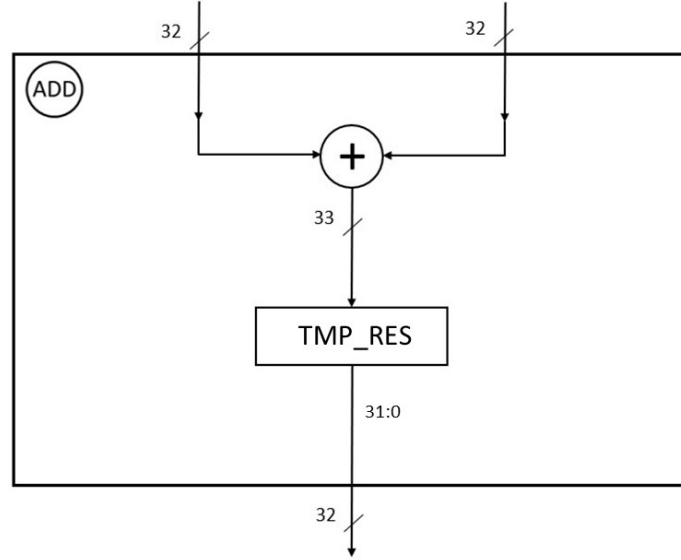


Figure 9: schematic of an addition between 32-bit operands

The custom addition takes as input two 32-bit operands and stores the result in a 33-bit temporary variable. This variable has a type *reg* and represents a logical node of a combinatorial network on which perform a combinatorial operation. 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 variable.

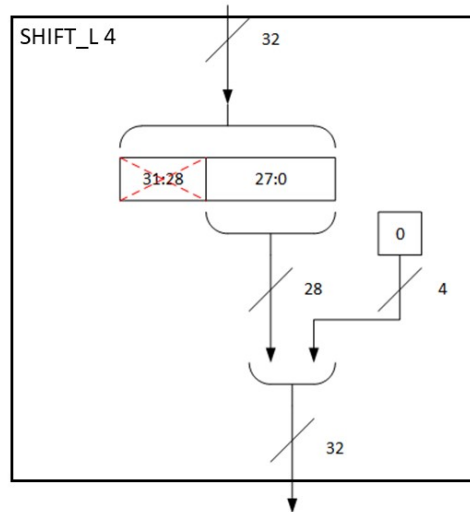


Figure 10: schematic of 4-bit left shift on a 32-bit operand

The custom 4-bit left shift takes as input a 32-bit value. From this value the 28 least significant bits are taken and then it is used the concatenation operator to insert 4 bits at zero, discarding the 4 most significant bits. 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.

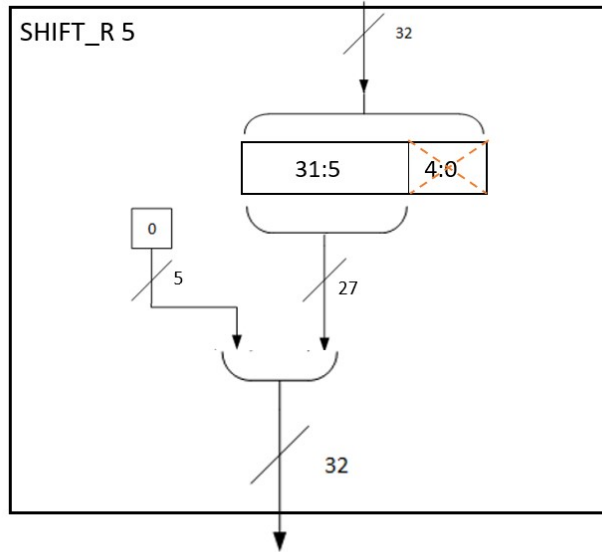


Figure 11: schematic of 5-bit right shift on a 32-bit operand

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.

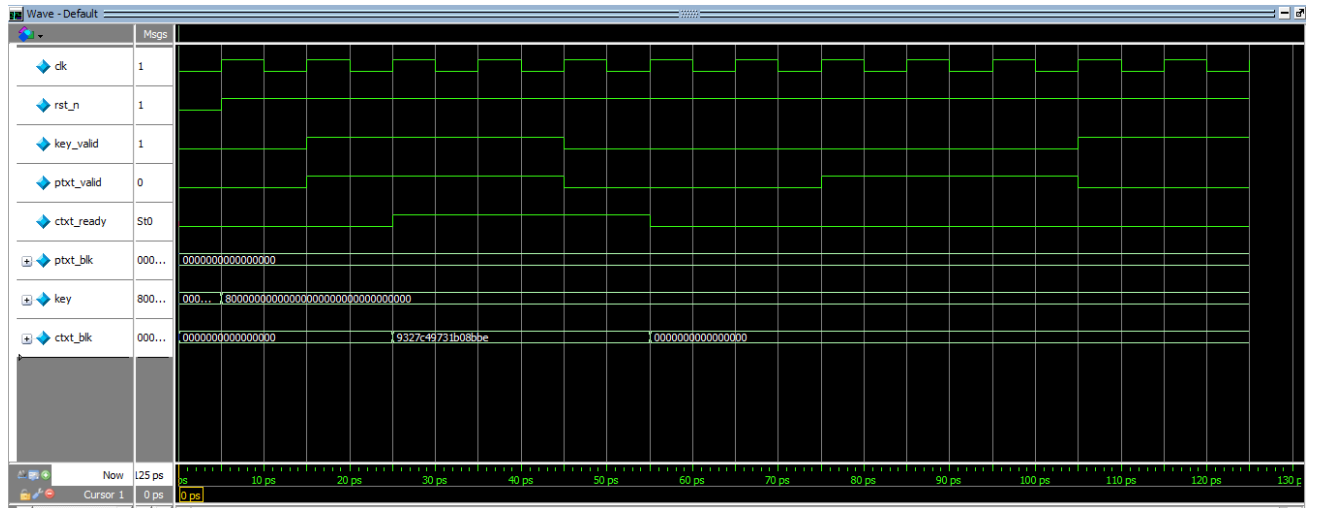


Figure 12: A waveform obtained from the corner cases' testbench.

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 *cbtxt_blk* and *cbtxt_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 *cbtxt_ready* from the previous ones. More in details, *cbtxt_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 *cbtxt*, the final result of the encryption process, is valid and available only when *cbtxt_ready* has a logic value to 1.

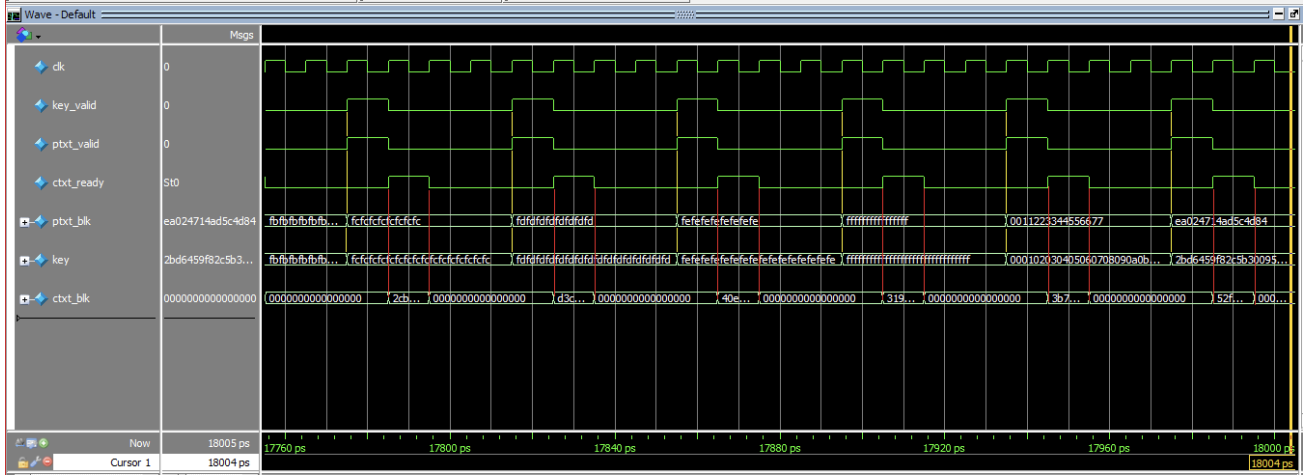


Figure 13: A waveform obtained from the testbench that checks the encryption

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*. Using red lines, Figure 13 highlights also 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, as expected.

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 according to all the possible combination of signals that *ptxt_valid* and *key_valid*, because only when both assume logic value 1 the encryption is performed, and signal *ctxt_ready* is set to logic value 1. Those two inputs are 1-bit signals, so in the testbench there are four different cases.

In this testbench a single plaintext and a single key from the test vector stimulate the module, because its behavior does not 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

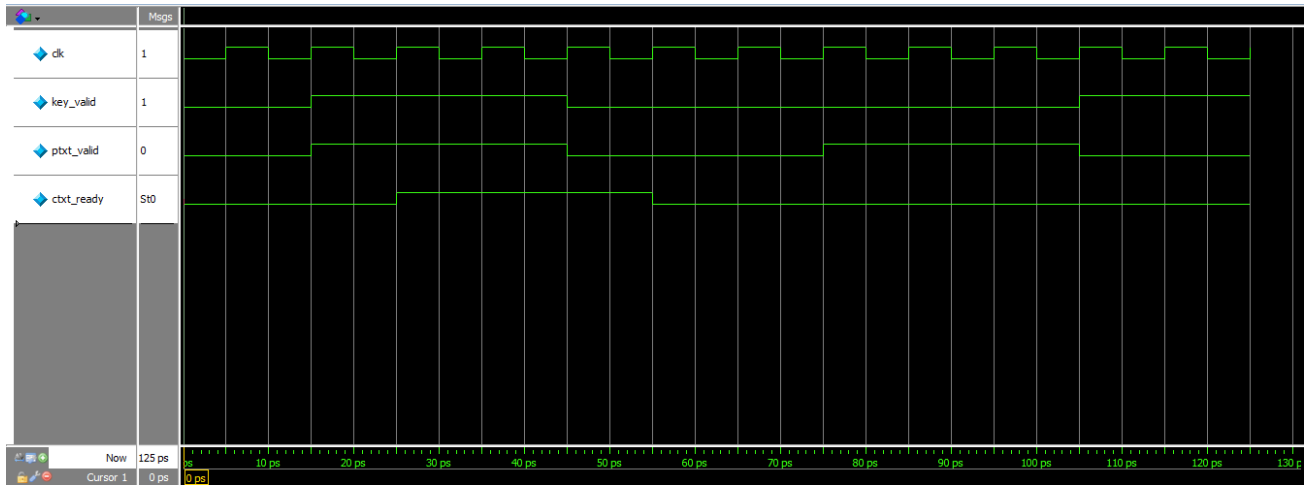


Figure 15: Waveform that shows the right behavior of signal *ctxt_ready*

Figure 15 shows that signal *ctxt_ready* is high (logic 1) if and only if both signals *ptxt_valid* and *key_valid* are at logic 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. In the portion of code above there are three clock event `@(posedge clk)`: at the first one *ptxt_valid* and *ctxt_valid* are set to logic value 1, at the second one these two signals are read and the encryption is triggered, performed and *ctxt_ready* is set to logic value 1. At the third clock event, it is finally possible to read the values of *ctxt_ready* and *ctxt*.

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 is 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 is 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 passed.

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

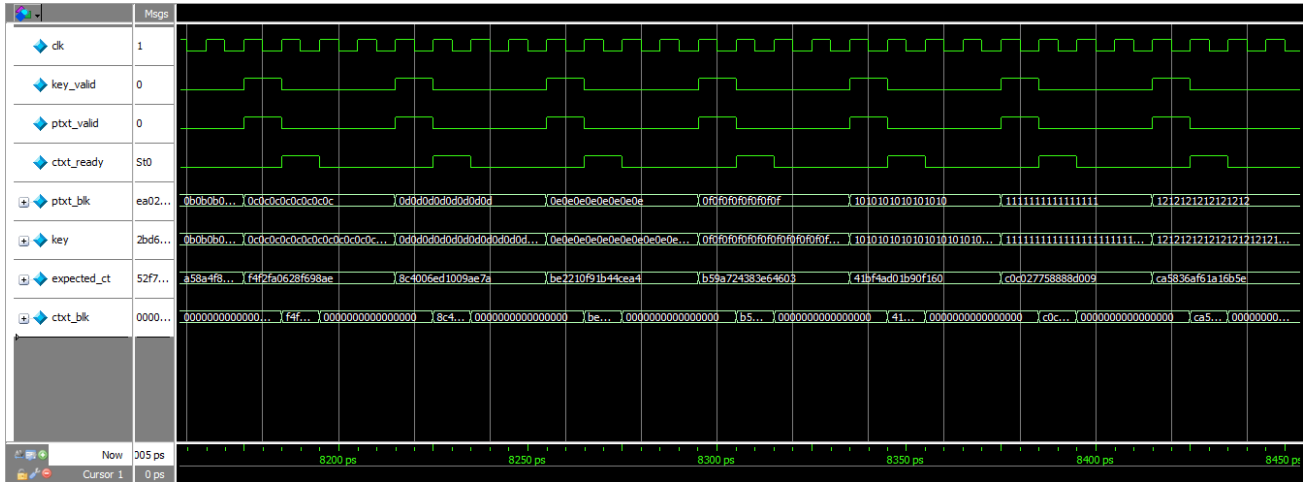


Figure 17: Waveform that shows the right behavior of the encryption module

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

        @(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. Before to compare the processed ciphertext and the expected one, the two validity signals are set to logic value 0 to avoid to perform the next encryption. After that, it is necessary to wait until *ctxt_ready* switches from 0 to 1 because is the moment the computed ciphertext is available and valid. 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. These reports show 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 is not 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 and in a real context it is highly likely that the TEA module is integrated into a more complex system, therefore the pins of TEA module would turn out to be internal nodes of the system. If the FPGA were used to implement only TEA module, then it would be necessary to use 261 of the pins, which make up the 70% of the FPGA's available pins.

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 time constraints about input and output signals of the implemented module, it has been written a *.sdc* file.

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: Content of *.sdc* file

In Figure 20 it is shown the *.sdc* file developed to create several time constraints. The command *create_clock* 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 command *set_false_path* says that the synthesis engine does not 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 commands, *set_input_delay* and *set_output_delay*, that represent 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. As shown also in Figure 18 and Figure 19, it is possible to see that all the signals except the clock are virtual, because it is highly likely that the TEA module is integrated in a more complex system. Report in Figure 21 shows that the clock signal has not 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

The previous summaries, obtained from the Static Timing Analysis process on Quartus, show 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. It has to be considered the lower frequency of 3.01 MHz because this is the frequency at which the module is able to work in the worst case.

As outlined from the results shown in Figure 22 and Figure 23, the maximum frequency obtained is 3.01 MHz. This is due to the fact that the chosen architectural approach implements all 32 rounds in cascade and performs all the operation of the 32 rounds in a single clock cycle. An alternative approach could implement a single round of encryption that it is used 32 times, so the module needs 32 clock cycles to perform a single encryption. Both approaches have their own characteristics concerning critical paths and latency.

Regarding critical paths, in the chosen architecture the critical path is equal to the critical path of 32 rounds, because they are in cascade, while in the other approach the critical path is the path of a single round. Concerning latency, the chosen architectural approach should have a latency lower than the one in the alternative approach: this is due to the fact that the chosen approach implements all 32 rounds in a single clock cycle, while the other one performs a single encryption in 32 clock cycles, one for each round.