UNIVERSITY OF PISA
MASTER'S DEGREE IN CYBERSECURITY

TINY ENCRYPTION ALGORITHM IMPLEMENTED IN VERILOG

COURSE HARDWARE AND EMBEDDED SECURITY

Author(s)
Nicola Lepore
Francesco Copelli

Academic year 2023/2024

List of Figures

List of Tables

Contents

CHAPTER 1

Project Specifications

Introduction

The objective of this project is to design and implement the Tiny Encryption Algorithm. The Tiny Encryption Algorithm (TEA) is a lightweight block cipher known for its simplicity and efficiency. It is designed to encrypt 64-bit data blocks using a 128-bit key. The algorithm's structure is straightforward, consisting of 64 rounds of operations, typically grouped in pairs called cycles, leading to 32 complete cycles. TEA's simplicity extends to its key schedule, which mixes the 128-bit key uniformly across all rounds. This report outlines the detailed specifications and analyzes the requirements for each phase of the project.

1.1 Project Development Stages and Specifications

The development of this project consists of several stages, each with specific requirements and analyses:

1.1.1 Creating a High-Level Model

The design and implementation of the TEA take place from the C code provided from the project specification and shown in Fig. 2.1

Analysis: The high-level model serves as the foundation for verifying the TEA algorithm's logic. It allows for initial validation and debugging in a more abstract environment, ensuring that the algorithm functions correctly before moving on to hardware design. For the full implementation, check Chapter 2

1.1.2 Writing the RTL Code

Description: Implement the Register Transfer Level (RTL) code for the Tiny Encryption Algorithm (TEA) module using SystemVerilog, see Chapter 3. This task involves converting the high-level algorithmic description into a synthesizable hardware description language (HDL) that is compatible with FPGA implementation.

Analysis: The process of RTL coding entails mapping the algorithmic logic to a hardware structure that captures the functional intent within the constraints of digital design. It is essential that the RTL accurately mirrors the algorithm's operations while adhering to design specifications and timing constraints required for FPGA synthesis.

Module Interface Specification:

```
module tiny_encryption_algorithm(
    input               clk,        // clock
    input               rst_n,      // asynchronous reset active low
    input               key_valid,  // 1 = input data stable and valid, 0 = otherwise
    input               ptxt_valid, // 1 = input data stable and valid, 0 = otherwise
    input       [63:0]  ptxt,       // plaintext
    input       [127:0] key,        // key
    output reg  [63:0]  ctxt,       // ciphertext
    output reg          ctxt_ready  // 1 = output data stable and valid, 0 otherwise
);
```

Listing 1.1: *RTL module code of TEA in Verilog*

The TEA module, shown in Listing 1.1 and represented in Fig. 1.1, defines the following ports:

- **clk:** This is the clock input that drives the sequential elements of the module.
- **rst_n:** An active-low asynchronous reset signal, allowing immediate reset of the module's state, independent of the clock cycle.
- **key_valid:** This input signal indicates the validity of the key data on the input bus. A high level (`1'b1`) indicates stable and valid key data, low level otherwise.
- **ptxt_valid:** Similar to `key_valid`, this signal asserts when the plaintext data on the input bus is stable and valid (`1'b1`), and deasserts (`1'b0`) when the plaintext data is not reliable.
- **ptxt:** A 64-bit bus carrying the plaintext data that is to be encrypted by the TEA function.
- **key:** A 128-bit bus providing the symmetric encryption key utilized by the TEA algorithm.
- **ctxt:** A 64-bit bus that outputs the encrypted ciphertext block processed by the TEA function.
- **ctxt_ready:** This signal indicates the availability of valid ciphertext on the output bus. A high level (`1'b1`) confirms that the output data is stable and ready for further processing, while a low level (`1'b0`) indicates that the ciphertext data is not yet valid.

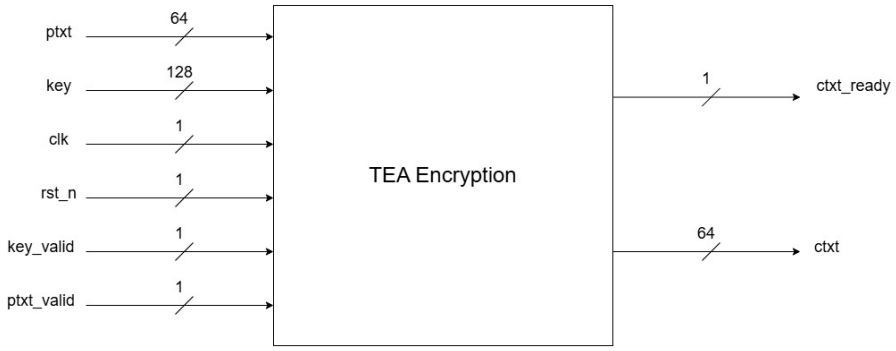

Figure 1.1: *TEA encryption module ports representation*

1.1.3 Writing Testbenches

Description: Create one or more testbenches in SystemVerilog to verify the functionality of the TEA module using ModelSim.

Analysis: Testbenches, contained in Chapter 4 & 5, are essential for validating the RTL code. They simulate various scenarios to ensure the TEA module operates as expected under different conditions. This step is critical for identifying and correcting any issues before hardware synthesis.

1.1.4 Performing Synthesis and Implementation

Description: Synthesize and implement the TEA module on the 5CGXFC9D6F27C7 FPGA device using Quartus, including Static Timing Analysis (STA) if necessary. This step has been covered in Chapter 6.

Analysis: Synthesis converts the RTL code into a gate-level representation suitable for FPGA implementation. Implementation involves placing and routing the design on the FPGA. The use of Quartus ensures that the design meets timing and resource constraints, and Static Timing Analysis (STA) helps in verifying that the design will function correctly at the desired clock speeds.

CHAPTER 2

High-level Model

As explained in Chapter 1.1, the project involves developing the encryption module for the Tiny Encryption Algorithm (TEA). Let's now take a closer look of the high-level model implementation showed in Fig. 2.1

```
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 2.1: *TEA encryption function code snippet.*

2.1 TEA Encryption Function Description

Function Declaration:

```
1 void encrypt (uint32_t v[2], const uint32_t k[4])
```

Listing 2.1: *TEA Code Declaration*

Parameters:

- `v`: An array of two 32-bit unsigned integers, `v[0]` and `v[1]`, representing the 64-bit data block to be encrypted, splitted in 2 parts of 32-bit each.

- `k`: An array of four 32-bit unsigned integers, `k[0]`, `k[1]`, `k[2]`, and `k[3]`, representing the 128-bit key, splitted equally in blocks of 32-bit each.

Local Variables:

```
int32_t v0=v[0], v1=v[1], sum=0, i;
int32 t delta=0x9E3779B9;
int32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];
```

Listing 2.2: *TEA Code Variable Initialization*

- `v0` and `v1`: Local copies of `v[0]` and `v[1]`, respectively.
- `delta`: A constant value `0x9E3779B9`, which is used in each round to prevent simple attacks based on round symmetry.
- `k0`, `k1`, `k2`, and `k3`: Local copies of `k[0]`, `k[1]`, `k[2]`, and `k[3]`, respectively.
- `sum`: A variable to accumulate the delta value over the rounds.
- `i`: Loop counter.

Encryption Process:

1. **Initialization:**
   - Copy the input values `v[0]` and `v[1]` into local variables `v0` and `v1`.
   - Initialize `sum` to 0.
   - Set `delta` to the value `0x9E3779B9`.
   - Copy the key values `k[0]`, `k[1]`, `k[2]`, and `k[3]` into local variables `k0`, `k1`, `k2`, and `k3`.
2. **Encryption Rounds:**

```
for (i = 0; i < 32; i++){
    sum += delta;
    v0 += ((v1 << 4) + k0) ^ (v1 + sum) ^ ((v1 >> 5))) + k1);
    v1 += ((v0 << 4) + k2) ^ (v0 + sum) ^ ((v0 >> 5))) + k3);
}
```

Listing 2.3: *Encryption Process Tea function*

   - Perform 32 iterations (rounds) of encryption.
   - In each round:
     - Increment `sum` by `delta`.
     - Update `v0` using the formula in the Eq. 1

$$v0+ = ((v1 \ll 4) + k0) \oplus (v1 + sum) \oplus ((v1 \gg 5) + k1); \quad (1)$$

     - Update `v1` using the formula in Eq. 2:

$$v1+ = ((v0 \ll 4) + k2) \oplus (v0 + sum) \oplus ((v0 \gg 5) + k3); \quad (2)$$

3. **Finalization:**
   - Store encrypted values back in the input array `v`:

```
v[0] = v0;
v[1] = v1;
```

Summary

The TEA encryption function takes a 64-bit plaintext block and a 128-bit key and performs 32 rounds of Feistel structure operations to produce a 64-bit ciphertext block. Each round involves a series of bit shifts, additions, and XOR operations, with a constant delta value to thwart attacks based on symmetry. The result is written back into the original input array, completing the encryption process.

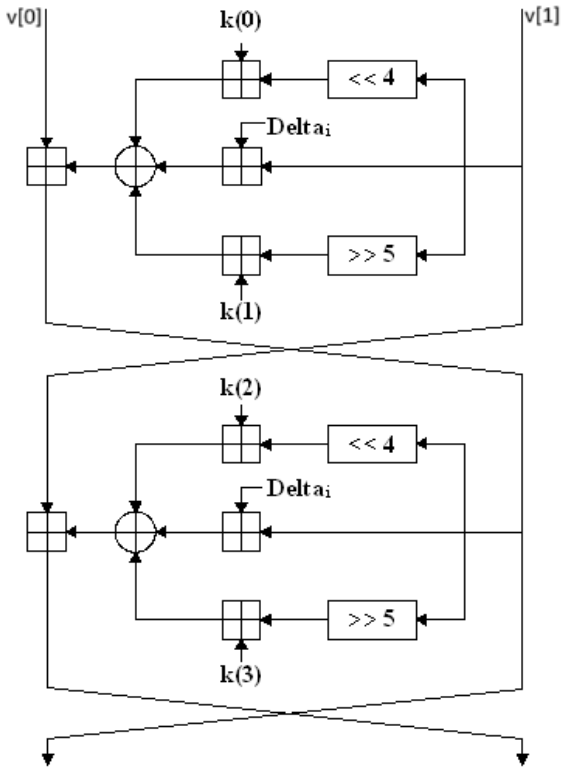

Figure 2.2: *Two Feistel rounds (one cycle) of TEA*

CHAPTER 3

RTL Design

The TEA_enc module, as an RTL design, is likely a component of a larger digital system. It would be implemented using an HDL and would describe the behaviour of a TEA encryption engine. The module would contain registers to store the encryption key, intermediate results, and control signals. Combinational logic would be used to perform the necessary arithmetic and logical operations for the encryption process.

The single iter-round-pipelined architecture has been used, in which the single round is characterised by the tea_round shown in Fig. 3.1.

The RTL design under consideration is centered around the implementation of the ‘tea_round‘ module, which implements a round of operations for a TEA-based encryption process, as depicted in Figure 3.1.

3.1 Overall Structure (`tea_round`)

3.1.1 Inputs

- `in`: A 64-bit input data block, divided into two 32-bit words.
- `key0`, `key1`, `key2`, `key3`: Four 32-bit key segments used in the encryption process.
- `sum`: A 32-bit value used in the round function.

3.1.2 Outputs

- `out`: A 64-bit output data block resulting from the round computation.

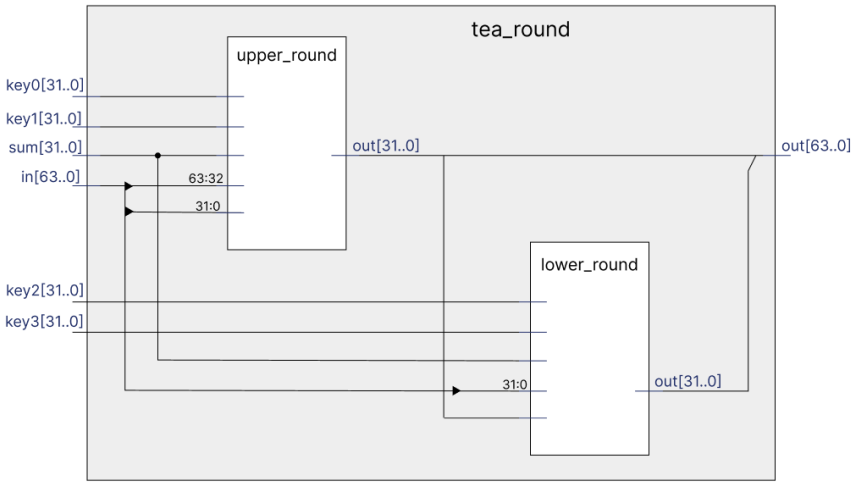

Figure 3.1: *tea_round representation*

3.1.3 Internal Wiring

- `upper_round_temp`: A 32-bit internal wire that holds the intermediate result produced by the `upper_round` submodule, as in Fig. 3.2.
- `lower_round_temp`: A 32-bit internal wire that holds the intermediate result produced by the `lower_round` submodule, as in Fig. 3.3.

These internal wires, `upper_round_temp` and `lower_round_temp`, are crucial for the overall function of the `tea_round` module. They temporarily store the results of the two sub-rounds (upper and lower), which are then concatenated to form the final 64-bit output.

3.2 Submodules

3.2.1 Upper Round (`upper_round`)

Inputs:

- `v0`: Upper 32-bit word of the input (`in[63:32]`).
- `v1`: Lower 32-bit word of the input (`in[31:0]`).
- `key0`, `key1`, `sum`: Key segments and sum for the computation.

Outputs:

- `out`: A 32-bit output that is passed to the lower round as part of the input.

Operation:

The `upper_round` submodule performs a series of arithmetic and logical operations on the input values to generate an intermediate 32-bit result:

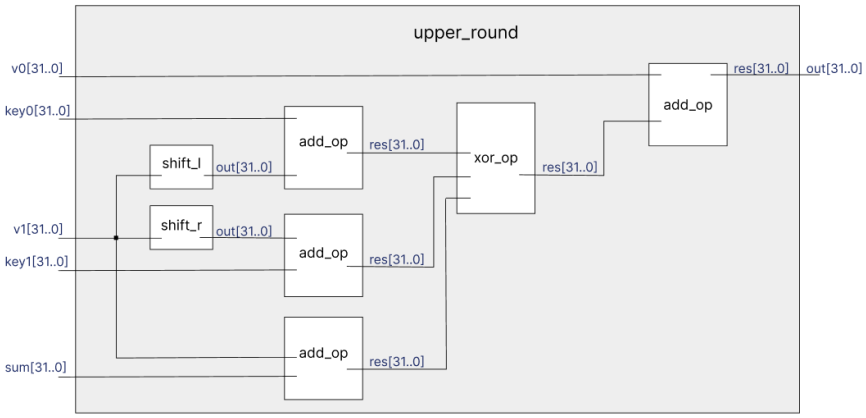

Figure 3.2: *Upper_round representation*

- The input `v1` is shifted left by 4 bits and right by 5 bits to produce two new 32-bit values.
- These shifted values are then added to `key0` and `key1`, respectively, generating two intermediate sums.
- A third sum is calculated by adding the original `v1` value to the `sum` input.
- All three sums are combined using a bitwise XOR operation to produce the final result.
- This final result is added to the `v0` input value, and the output is stored in `upper_round_temp`.

3.2.2 Lower Round (`lower_round`)

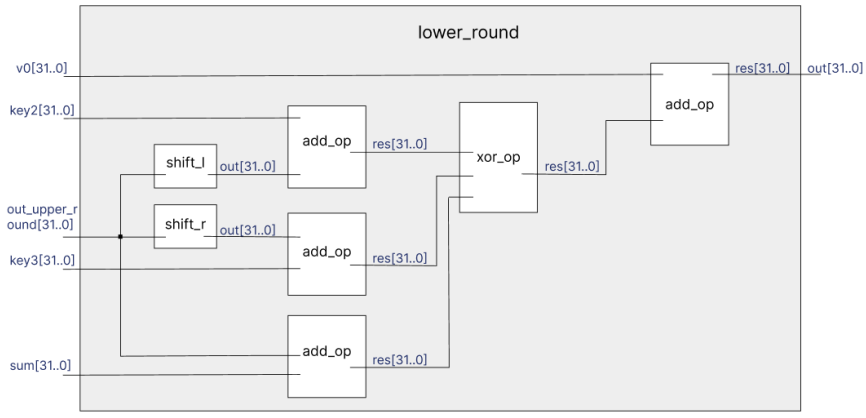

Figure 3.3: *Lower_round representation*

Inputs:

- `v0`: Lower 32-bit word of the input (`in[31:0]`).

- `v1`: Output from the `upper_round` (`upper_round_temp`).
- `key2`, `key3`, `sum`: Key segments and sum for the computation.

Outputs:

- `out`: A 32-bit output that, together with `upper_round_temp`, forms the final 64-bit output of the `tea_round`.

Operation:

The `lower_round` submodule mirrors the operations of the `upper_round`:

- The input `v1` (which is the output from the `upper_round`) is shifted left by 4 bits and shifted right by 5 bits, using respectively `shift_l` and `shift_r` as in Fig. 3.4, to generate two new values.
- These shifted values are added to `key2` and `key3`, respectively, producing two sums.
- The original `v1` is added to the `sum` input, forming a third sum.
- All three sums are combined using a bitwise XOR operation, as in Fig. 3.6, to produce the final 32-bit result.
- This result is added to the `v0` input value, and the output is stored in `lower_round_temp`.

3.3 Shift and Arithmetic Operations

3.3.1 `shift_l` and `shift_r`

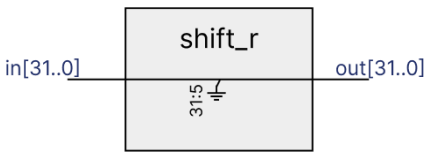

Figure 3.4: *Right shift representation*

- Perform left and right shifts on the input `v1` by 4 and 5 bits, respectively.

3.3.2 `add_op`

- Performs addition operations between two 32-bit inputs.

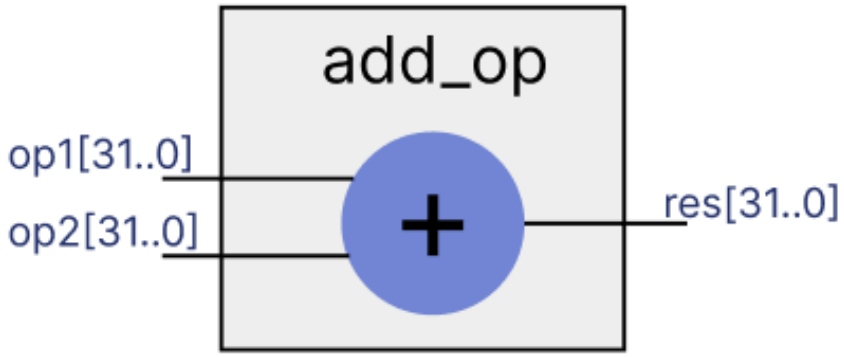

Figure 3.5: *Add operation representation*

3.3.3 `xor_op`

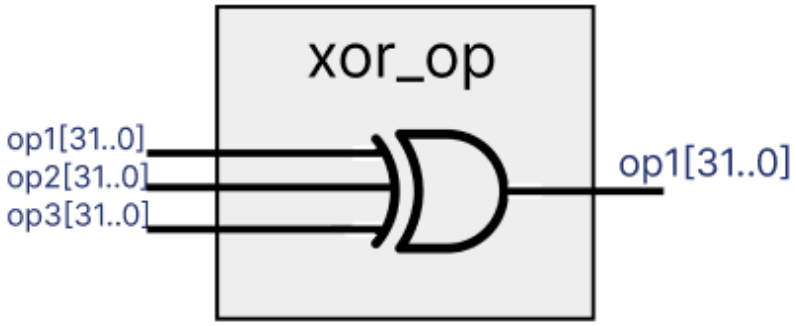

Figure 3.6: *XOR ternarium representation*

- Performs a bitwise XOR operation, in Fig. 3.6, across three 32-bit inputs.

Final Output:

The 64-bit `out` is formed by concatenating the outputs from the `upper_round` and `lower_round` (`upper_round_temp` and `lower_round_temp`).

3.4 FSM Implementation and State Register

The Finite-State Machine (FSM) is a crucial component of the system, responsible for controlling the sequence of operations and transitions between different states. Its functionality is underpinned by a dedicated register, termed the STAR (STAtus Register). The STAR maintains the current state of the FSM, dictating the subsequent actions to be executed and guiding the overall flow of the system's modules.

The FSM is implemented as an `always` block, conditioned on two events:

1. **Positive Edge of the System Clock:** This triggers the state transition and execution of the current state's logic.
2. **Negative Edge of the Asynchronous Reset:** This condition initiates the system's reset procedure, forcing the FSM to return to a predefined initial state. This allows

for a controlled and predictable initialization phase, ensuring all relevant registers are set to their appropriate values.

The code snippet in Listing 3.1 is the equivalent code in Fig. 2.1

```
always @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
        key0 <= 32'b0;
        key1 <= 32'b0;
        key2 <= 32'b0;
        key3 <= 32'b0;
        sum <= 32'b0;
        in <= 64'b0;
        round <= 0;
        ctxt_ready_temp <= 1'b0;
        out <= 64'b0;
        star <= s0;
  end else begin
    // State transition and operation logic based on STAR (OMISSIS)
  end
end
```

Listing 3.1: *Core of the TEA implementation*

The `else` branch implements the FSM with the following states:

- **State S0**: This is the initial state. If both signals `key_valid` and `ptxt_valid` are asserted (i.e., equal to 1), the FSM transitions to State S1; otherwise, the FSM remains in State S0. In this state, no further operations are performed beyond checking the conditions for state transition.
- **State S1**: In this state, the FSM performs the initialization of the registers. The `round` counter is set to 0, and the registers `key0`, `key1`, `key2`, and `key3` are loaded with their respective 32-bit segments of the `key` input. The `sum` register is set to `DELTA`. The `ctxt_ready_temp` signal is reset (set to 0), and the `out` register is set to 64 bits of zeros. The `in` register is loaded with the value of `ptxt`. After these operations, the FSM transitions to State S2.
- **State S2**: In this state, the `in` register is updated with the value of `round_output`, and the `sum` register is incremented by `DELTA`. If the `round` counter reaches the value 32, the `ctxt_ready_temp` signal is set to 1, indicating that the output is ready, and the FSM returns to State S0. Additionally, the `out` register is updated with the value of `in`. If the `round` counter has not yet reached 32, it is simply incremented by 1, and the FSM remains in State S2.
- **Default**: In any undefined case, the FSM returns to State S0, ensuring a safe fallback to the initial state.

The state diagram shown in Fig. 3.7 represent the state explained above.
The relationships between each state, shown previously in Fig. 3.7 are described in the table in Fig. 3.8

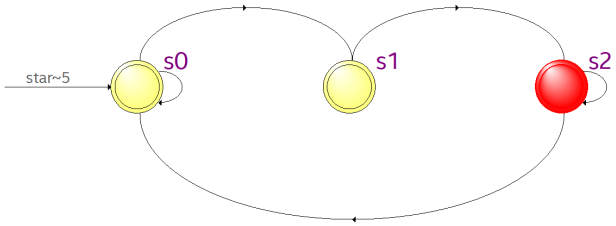

Figure 3.7: *FSM representation*

	Source State	Destination State	Condition
1	s0	s1	(key_valid).(ptxt_valid)
2	s0	s0	(!key_valid) + (key_valid).(!ptxt_valid)
3	s1	s2	
4	s2	s2	(!ctxt_ready_temp)
5	s2	s0	(ctxt_ready_temp)

Figure 3.8: *Status Change Conditions Table*

CHAPTER 4

Interface Specifications and Expected Behavior

4.1 Expected Behaviour and Example Scenarios

This section presents several use cases, illustrating the expected behaviour of the module through waveforms.

4.1.1 Main Use Case: Valid Key and Plaintext

In the main use case, a valid key and plaintext are provided, and the module successfully encrypts the data, as shown in Fig. 4.1. The following steps describe the process:

1. The reset signal has been asserted due to be sure that all the signals are restored to initial values and then after the deassertion, the plaintext (`ptxt`) and key (`key`) are provided, with the validity signals (`key_valid` and `ptxt_valid`) asserted.

2. The module processes the inputs and, after 35 clock cycles, asserts `ctxt_ready` to indicate that the ciphertext is available.

3. The user can then read the ciphertext output (`ctxt`), until the reset signal is asserted.

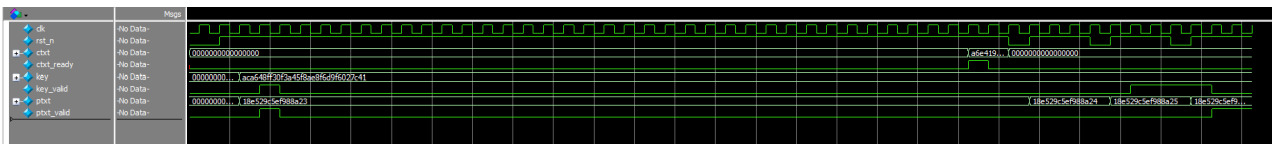

Figure 4.1: *Waveform for the main use case: valid key and plaintext.*

4.1.2 Corner Case: Delayed Validity Signal

In this scenario, the validity signals for the key or plaintext are delayed, as shown in Fig. 4.2. The module waits until both `key_valid` and `ptxt_valid` are asserted before starting the encryption process.

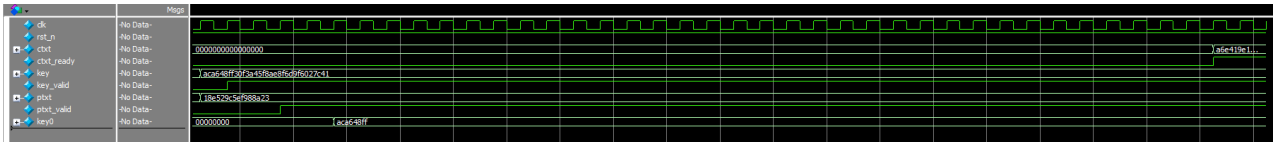

Figure 4.2: *Waveform for the corner case: delayed validity signal.*

4.1.3 Error Case: Premature Input Change

If the plaintext or key inputs change before the encryption process is complete, and the validity signals are not asserted, the output may be incorrect, as shown in Fig. 4.3. This case illustrates the importance of ensuring input stability before asserting the validity signals.

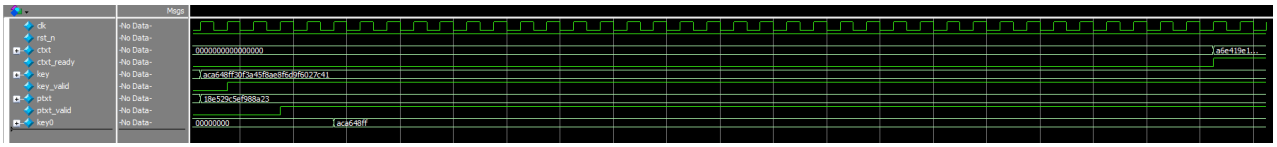

Figure 4.3: *Waveform for the error case: premature input change.*

CHAPTER 5

Functional Verification

The goal of functional verification is to ensure that the module operates according to the expected behaviour described in Chapter 4. Two testbenches were created to validate different aspects of the module's functionality.

5.1 Testbench for Functional Verification of Encryption Process

The testbench, called `tb_tea_functional`, was created to verify that the module correctly encrypts the data. This accepts inputs from three files: `keys.txt`, `pt.txt` (plaintext), and `ct.txt` (ciphertext). These files were generated using a Python script (reported in Chapter 5.2), ensuring consistency and correctness in the test vectors.

5.1.1 Testbench Architecture

The architecture of `tb_tea_functional` is more comprehensive and includes:

- A clock signal (`clk`) and reset signal (`rst_n`);
- Registers for the plaintext, key, and expected ciphertext, along with their corresponding validity signals.
- The `tiny_encryption_algorithm` module, instantiated within the testbench.
- File descriptors for reading the `keys.txt`, `pt.txt`, and `ct.txt` files.

5.1.2 Test Operation

The operation of `tb_tea_functional` involves the following steps:

- The testbench reads a key, plaintext, and the expected ciphertext from the respective files.
- These values are provided to the module, with the `key_valid` and `ptxt_valid` signals being asserted to indicate that the inputs are stable and valid.
- The module processes the input values and generates the output ciphertext.
- The testbench compares the generated ciphertext with the expected value read before from the `ct.txt` file.
- A message is displayed for each test case, indicating whether the generated ciphertext matches the expected value or not.
- The test continues until all test vectors have been processed, after which the simulation ends.

This testbench thoroughly verifies the correctness of the encryption process, ensuring that the `tiny_encryption_algorithm` module produces accurate results across a range of input vectors.

5.2 Python Algorithm for functional Testing

To check the consistency and the correctness of the algorithm and testbenches, a python script has been created and listed in Listing 5.1. The script is composed by:

- **tea_encrypt**, it's the main core of the script, it takes a plaintext and a key and perform the tiny_encryption_algorithm on them, by dividing the plaintext in two parts (v0, v1) and the keys in 4 parts stored in an array. After that, 32 rounds are performed to obtain the partial ciphertext. The 64 bit return ciphertext is composed of two parts (v0, v1) of 32 bits each.
- **generate_random_plaintexts**, generates *num_tests* random plaintext
- **generate_random_keys** generates *num_tests* random keys
- **generate_expected_ciphertexts** generates *num_tests* random ciphertexts, performing the tea_algorithm function giving as input the pair plaintext, ciphertext taken from the previous generation(s).

In the main function, we state how many tests we want to generate and the function calls all the generation functions due to perform the task.

```
1 import random
2
3 DELTA = 0x9e3779b9
4
5 def tea_encrypt(ptxt, key):
6     """ Esegue TEA su un blocco di testo in chiaro usando una chiave a 128 bit """
7     v0, v1 = (ptxt >> 32) & 0xFFFFFFFF, ptxt & 0xFFFFFFFF
8     key_parts = [(key >> shift) & 0xFFFFFFFF for shift in (96, 64, 32, 0)]
9     sum = 0
```

```
    for _ in range(32):
        sum = (sum + DELTA) & 0xFFFFFFFF
        v0 = (v0 + (((v1 << 4) + key_parts[0]) ^ (v1 + sum) ^ ((v1 >> 5) + key_parts
    [1]))) & 0xFFFFFFFF
        v1 = (v1 + (((v0 << 4) + key_parts[2]) ^ (v0 + sum) ^ ((v0 >> 5) + key_parts
    [3]))) & 0xFFFFFFFF
    ctxt = (v0 << 32) | v1
    return ctxt

def generate_random_plaintexts(num_tests):
    f""" Genera un file contenente {num_tests} plaintext casuali """
    with open("pt.txt", "w") as f:
        for _ in range(num_tests):
            ptxt = random.getrandbits(64)
            f.write(f"{ptxt:016X}\n")

def generate_random_keys(num_tests):
    f""" Genera un file contenente {num_tests} chiavi casuali """
    with open("keys.txt", "w") as f:
        for _ in range(num_tests):
            key = random.getrandbits(128)
            f.write(f"{key:032X}\n")

def generate_expected_ciphertexts(num_tests):
    f"""Genera un file con {num_tests} ciphertext attesi per ogni coppia di plaintext
    e chiave """
    with open("pt.txt", "r") as f_ptxt, open("keys.txt", "r") as f_key, open("ct.txt",
     "w") as f_ctxt:
        for ptxt_line, key_line in zip(f_ptxt, f_key):
            ptxt = int(ptxt_line.strip(), 16)
            key = int(key_line.strip(), 16)
            ctxt = tea_encrypt(ptxt, key)
            f_ctxt.write(f"{ctxt:016X}\n")

num_tests = 500 # Numero di test da generare

generate_random_plaintexts(num_tests)
generate_random_keys(num_tests)
generate_expected_ciphertexts(num_tests)
print(f"Generati {num_tests} test di plaintext, chiavi e ciphertext attesi.")
print("#" * 80)
```

Listing 5.1: *Python scirpt functional verification*

CHAPTER 6

FPGA Implementation Results

The device chosen for FPGA implementation is Cyclone V (5CGXFC9D6F27C7). The results are summarized, including resource utilization, total registers, total pins, and other metrics. Special attention is paid to the Static Timing Analysis (STA), where five tests were performed to determine the maximum operating frequency.

6.1 Implementation Details

Fitter Status	Successful - Wed Sep 4 17:03:31 2024
Quartus Prime Version	23.1std.0 Build 991 11/28/2023 SC 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)	422 / 113,560 (< 1 %)
Total registers	374
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 6.1: *FPGA Implementation Summary for* `tiny_encryption_algorithm`

The implementation process was carried out using Intel Quartus Prime Version 23.1std.0, as shown in Figure 6.1. The target FPGA device is suitable for low-power and high-performance applications. The design was successfully compiled, and the resource utilization metrics are as follows:

- **Logic utilization (in ALMs):** 422 / 113,560 (<1%)
- **Total registers:** 374
- **Total pins:** 1 / 378 (<1%)
- **Total virtual pins:** 260
- **Total block memory bits:** 0 / 12,492,800 (0%)
- **Total DSP Blocks:** 0 / 342 (0%)

6.2 Static Timing Analysis (STA)

The maximum operating frequency of the design was determined through Static Timing Analysis (STA). STA is crucial in identifying timing constraints and ensuring that the design operates correctly at the desired frequency. Five STA tests were conducted:

Table 6.1: *Table of Static Timing Analysis using different frequencies*

Test Number	Frequency	Slow 110mV 85C Model	Slow 110mV 0C Model	Result
Test 1	76.92MHz	87.88MHz	84.42MHz	OK
Test 2	80.00MHz	85.08MHz	83.49MHz	OK
Test 3 (Ideal)	83.33MHz	85.32MHz	83.84MHz	OK
Test 4	86.96MHz	82.06MHz	80.65MHz	FAIL
Test 5	90.91MHz	86.51MHz	84.41MHz	FAIL

1. **Test 1 and 2:** Tests performed below the ideal frequency in order to verify that Test 3 is correct.
2. **Test 3 (Central):** ideal test, where the frequency was perfectly balanced to meet the design's timing requirements without any errors. This frequency is considered the maximum safe operating frequency.
3. **Test 4 and 5:** Tests conducted at a slightly higher frequency than ideal, which caused errors.

The results of these STA tests indicate that while the design can tolerate frequencies slightly above the ideal, the most stable and reliable operation is achieved at the frequency determined in Test 3.

6.2.1 Latency and Throughput

To calculate the latency given that the output takes 35 clock cycles at a frequency of 83.33 MHz,the formula used is Eq. 1.

$$Latency = \frac{\#\ cycles}{Clock\ frequency} \tag{1}$$

Substituting into Eq. 1 the values of clock cycles and clock frequencies produces the Eq. 2:

$$Latency = \frac{35}{83.33 \times 10^6} seconds \approx 4.2 \times 10^{-7}\ s = 420ns \tag{2}$$

To calculate the throughput, the formula used is Eq. 3:

$$Throughput = \frac{\#\ bits\ output}{Latency} \tag{3}$$

Knowing that, the latency, obtained from Eq. 2 is 4.2×10^{-7} seconds, and each output computes 64 bits, we substitute these values into equation 3, obtaining equation 4.

$$Throughput = \frac{64\ bits}{4.2 \times 10^{-7}\ seconds} \approx 1.52 \times 10^8 \text{ bits per second} = 152 \text{ Mbps} \tag{4}$$

Therefore, we found that the latency, from Eq. 2 is approximately $420ns$, and instead the throughput is calculated from Eq. 4, is approximately 152 Mbps.

6.3 Conclusion

The FPGA implementation of the `tiny_encryption_algorithm` was successful, with minimal resource utilization and no block memory or DSP blocks required. The STA tests confirmed that the design operates reliably at the ideal frequency determined in Test 3, with some tolerance for higher frequencies. These results suggest that the design is well suited for implementation on the Cyclone V FPGA for encryption tasks.