UNIVERSITY OF PISA
MASTER'S DEGREE IN CYBERSECURITY

STREAM CIPHER V5

COURSE HARDWARE AND EMBEDDED SECURITY

Author(s)
Leonardo Corsini

Academic year 2024/2025

Contents

CHAPTER 1

Project Specifications

This project aims to design and implement a stream cipher based on a Galois Multiplication function of the AES algorithm that supports both encryption and decryption. Stream ciphers encrypt data byte-by-byte and are often used in scenarios where a continuous flow of data must be processed in real time. For this reason, throughput is a critical metric: a design capable of sustaining one processed byte per cycle avoids bottlenecks in high-speed communication systems or embedded systems. So the design focuses on maximizing the throughput and the maximum clock frequency with a reasonable latency.

1.1 Algorithm overview

The design accepts plaintexts of any length, processes each plaintext byte at a time, and produces the ciphertext byte by byte. For each plaintext byte, the module will produce the corresponding ciphertext byte following this equation:

$$C[i] = P[i] \oplus S(CB[i]) \tag{1.1}$$

where

- $\oplus$ is the bitwise XOR operator.
- C[i] is the ith byte of the ciphertext.
- P[i] is the ith byte of the plaintext.
- CB[i] is the 32-bit value of the ith counter (counter block), for i = 0,1,2, ..., and it can be represented by the formula $CB[i] = K + i \mod 2^{32}$, being K the 32-bit symmetric (encryption/decryption) key and *mod* the modulo operation.

- S() is the Galois multiplication function of AES, taking a 32-bit vector A as input and calculating the output by doing this operation:

$$f = S(CB[i]) = xtime(A[1] \oplus A[2]) \oplus A[2] \oplus A[3] \oplus A[0] \tag{1.2}$$

  Where A[0], A[1], A[2], A[3] are the bytes of A from the most significant to the least significant one, and *xtime* is a function that takes as input a byte *d* and produces as output a byte *e* in this way:

$$e = \{d[6], d[5], d[4], d[3] \oplus d[7], d[2] \oplus d[7], d[1], d[0] \oplus d[7], d[7]\} \tag{1.3}$$

1.2 Implementation overview

To guarantee high throughput, the design implements a pipeline structure, allowing the processing of more than one byte per clock cycle, and the path separation with registers helps achieve a higher *fMAX* and data rate.

1.2.1 Gold model

A Python implementation was developed to verify the correctness of the encryption/decryption algorithm before the hardware implementation, and to produce test vectors for later design testing.

1.2.2 Hardware realization

The hardware design was developed in SystemVerilog and synthesized with the Quartus program in the Intel Cyclone V (5CGXFC9D6F27C7) board. The design implements a signal control logic, providing mechanisms to detect new messages, valid and stable input and output, and supports an asynchronous reset. In particular, with this kind of management, the stream cipher can process data for plaintext of any length. Some testbenches are also provided to guarantee the correctness of the implementation.

CHAPTER 2

High-level Model

To have a reference for the hardware implementation, the high-level model is implemented in Python as a script to generate test vectors.

2.1 Stream_cipher() function

The function *stream_cipher()* takes as input the 32-bit key, the byte to encrypt/decrypt, and the value *i*, which is the counter value that represents the position of the byte in the input inside the plaintext. It is summed to the *key* value to obtain the *counter_block* value for the specific byte. Note that in the *counter_block* assignment in the code in Figure 2.1, the operation $(key + i)$ & $0xFFFFFFFF$ needs to align the value generated in the high-level code to the RTL implementation, since in hardware, we will have overflow after the counter reaches the value $0xFFFFFFFF$, so it will start again from $0x00000000$ in the next iteration.

```
def stream_cipher(key: int, plaintext_byte: int, i: int) -> int:

    counter_block = (key + i) & 0xFFFFFFFF
    s_output = s(counter_block)
    ciphertext_byte = (plaintext_byte & 0xFF) ^ s_output

    return ciphertext_byte
```

Figure 2.1: *Code of the function stream_cipher().*

2.2 S() function

After the *counter_block* calculation, its value is passed as input to the s() function.

```
def s(A: int) -> int:
    A &= 0xFFFFFFFF

    A0 = (A >> 24) & 0xFF
    A1 = (A >> 16) & 0xFF
    A2 = (A >> 8)  & 0xFF
    A3 = (A >> 0)  & 0xFF

    xtime_output = xtime(A1 ^ A2)

    f = xtime_output ^ A2 ^ A3 ^ A0

    return f & 0xFF
```

Figure 2.2: *Code of the function s().*

This function subdivides the *counter_block* value in 4 bytes as described in Chapter 1. After calculating the *xtime* value, it produces the f result by doing the *XOR* operation between *xtime_output*, *A2*, *A3*, and *A0*.

2.3 Xtime() function

The *xtime()* function takes as input a byte and it produces as output another byte, appropriately created as in Chapter 1.

```
def xtime(d: int) -> int:
    d &= 0xFF

    d7 = (d >> 7) & 1
    d6 = (d >> 6) & 1
    d5 = (d >> 5) & 1
    d4 = (d >> 4) & 1
    d3 = (d >> 3) & 1
    d2 = (d >> 2) & 1
    d1 = (d >> 1) & 1
    d0 = (d >> 0) & 1

    e7 = d6
    e6 = d5
    e5 = d4
    e4 = d3 ^ d7
    e3 = d2 ^ d7
    e2 = d1
    e1 = d0 ^ d7
    e0 = d7

    e = (e7 << 7) | (e6 << 6) | (e5 << 5) | (e4 << 4) | \
        (e3 << 3) | (e2 << 2) | (e1 << 1) | e0

    return e
```

Figure 2.3: *Code of the function xtime().*

2.4 Test vectors generator

The test vector generator script implements the algorithm, creates a new key for each new plaintext, and gives as input to the *stream_cipher()* function the value *i* to update the *counter_block*. After the entire ciphertext is generated, it writes the key, the plaintext, and the ciphertext to 3 files with .mem extension, to be read by the testbench in SystemVerilog. Every plaintext and key is generated randomly using the urandom function, and the entire operation is then repeated 10 times to have a large number of test vectors.

```python
from stream_chipher_high_level import stream_cipher
from os import urandom
from random import randint



for i in range(0,10):
    ciphertext_bytes = bytearray()
    plaintext_bytes = bytearray()

    key = urandom(4)
    plaintext = urandom(8)
    for j in range(0,8):
        p_byte = plaintext[j]
        key_32bit = int.from_bytes(key, byteorder='big')
        c_byte = stream_cipher(key_32bit, p_byte, j)
        ciphertext_bytes.append(c_byte)
        plaintext_bytes.append(p_byte)

    with open("../modelsim/tv/keys.mem", "a") as f:
        f.write(f"{key.hex()}\n")

    with open("../modelsim/tv/plaintexts.mem", "a") as f:
        f.write(f"{plaintext_bytes.hex()} \n")

    with open("../modelsim/tv/ciphers.mem", "a") as f:
        f.write(f"{ciphertext_bytes.hex()}\n")
```

Figure 2.4: *Code for the test vectors generator script.*

CHAPTER 3

RTL Design

The RTL Design implements the stream cipher with a 4-stage pipeline controlled by a Moore FSM in System Verilog.

3.1 Design choice

Since Stream Ciphers are typically used in real-time contexts, a high data rate is crucial.

The chosen architecture is based on a 4-stage pipeline that separates counter update, keystream derivation through the S() function, and the final XOR stage.

This provides two main advantages:

- The critical combinatorial path is shortened, allowing a higher maximum clock frequency compared to a full combinatorial architecture.
- After the pipeline is filled, the design sustains a throughput of one byte per cycle, which is desirable for stream ciphers where long messages must be processed with minimal latency.

Alternative architectures, such as fully combinatorial or multi-cycle designs, were discarded because they either limit the achievable fMAX or the throughput.

The pipeline introduces a latency of 4 clock cycles, which is an acceptable trade-off for processing messages of any length with a high throughput.

The initial choice was a 3-stage pipeline, but during implementation, I realized that a 4-stage pipeline would be better.

In Chapter 6, the details of this choice are explained.

It's also important to note that this design can instantly re-key without the use of the reset pin, which improves usability and can help in obtaining a higher data rate when more different messages are passed.

3.2 FSM architecture

The control FSM machine is responsible for coordinating state transitions and ensuring correct stage changes in the pipeline. The state machine was intentionally reduced to only two states because the pipeline processes multiple stages concurrently, and no additional control complexity is required to manage its flow. This minimalist approach avoids unnecessary control overhead while maintaining correct synchronization across the pipeline stages. The two states are: IDLE and PIPELINE_STATE:

1. **IDLE:** In this state, the machine waits for a new message and a valid input. When such data is received, the state will change to PIPELINE_STATE.
2. **PIPELINE_STATE:** This is the actual working state, where data encryption and decryption are handled using a pipeline. Each stage implements a part of the whole algorithm. In the first stage, the counter_block value is incremented; in the second, the function *S()* is computed; in the third, the final XOR is performed; and finally, the data is presented to the output registers.

   At every stage, the valid_in and encrypt_in signals propagate, ensuring the correctness of the data analyzed. The state will become IDLE again when no data is being processed inside the pipeline, and there are no new messages. Each pipeline stage includes its own valid and encrypt registers, which propagate the metadata alongside the data path. This avoids misalignment between the processed byte and its associated flags and ensures that output timing is fully deterministic. This propagation mechanism also allows the pipeline to correctly handle sparse data arrivals, where *valid_in* does not assert in consecutive cycles.

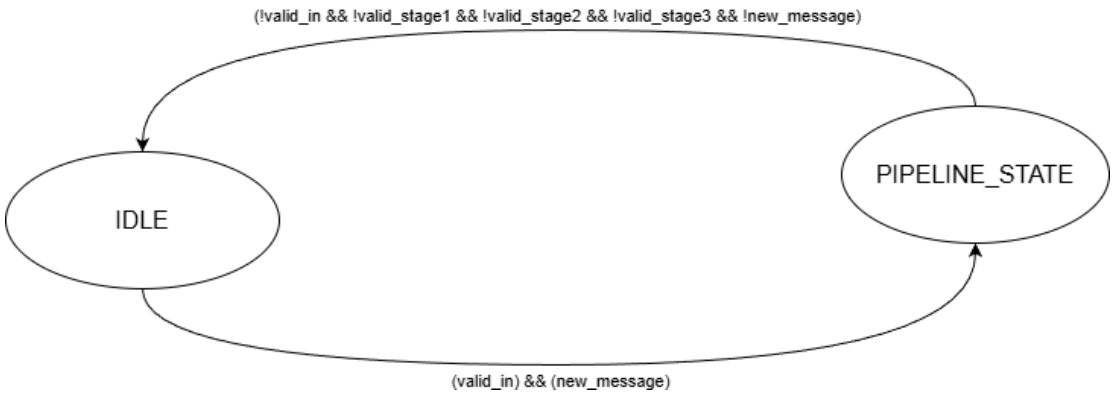

Figure 3.1: *Representation of the FSM states and transitions.*

3.3 High-level Block Diagram

The Figure 3.2 represents the High-level block diagram of the Project. It shows the pipeline structure of the *stream_cipher* module (the top-level of the project). *Counter block*, *s_function*, and *XOR* are combinatorial modules responsible for performing the processing. At the same time, the stage transition of the data resulting from each operation is managed by stage registers, which are synchronous components, as is the FSM control logic.

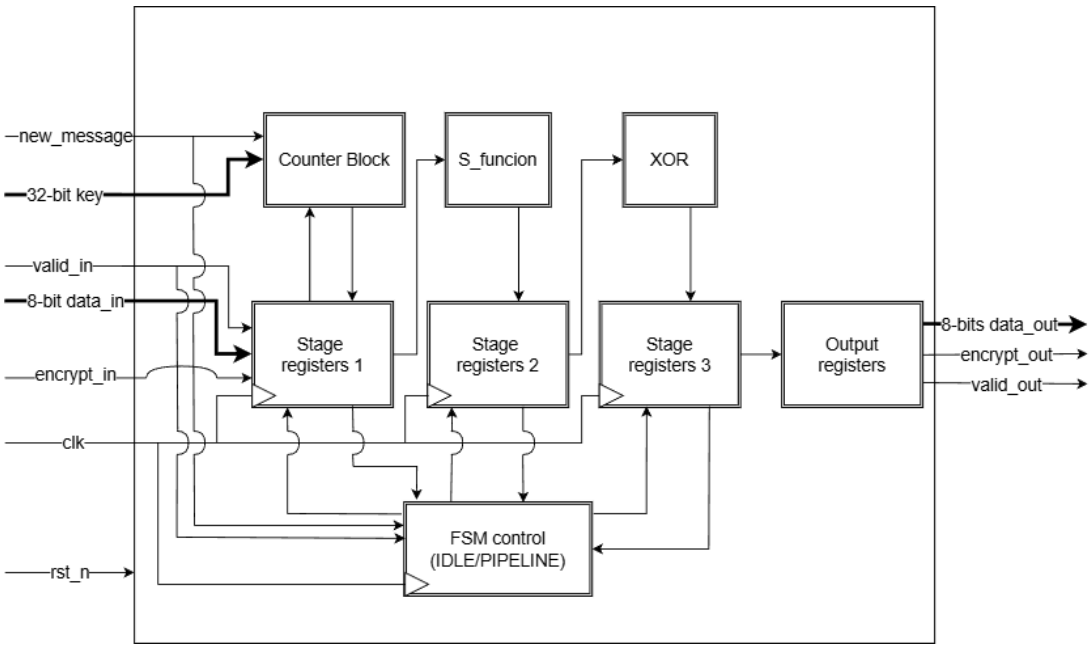

Figure 3.2: *High-level block diagram.*

CHAPTER 4

Interface Specifications and Expected Behavior

4.1 Top level interface

4.1.1 Inputs:

- ***new_message***: It is a single-bit input that signals when a new message arrives and allows the *counter_block* module to set the input key as the initial value of the counter.
- ***key***: It is a 32-bit input containing the key value for the encryption/decryption operations.
- ***valid_in***: It is a single-bit input that indicates whether the 8-bit data to encrypt/decrypt is valid or not.
- ***data_in***: It is an 8-bit input containing the byte of the plaintext/ciphertext to encrypt/decrypt.
- ***encrypt_in***: It is a single-bit input that indicates whether the byte to process has to be encrypted or decrypted. This does not affect the computation, but it is propagated to maintain metadata continuity.
- ***clk***: This is the main system clock.
- ***rst_n***: This is the active low asynchronous reset.

4.1.2 Outputs:

- ***data_out***: This is the 8-bit output resulting from the encryption/decryption operation.

- ***encrypt_out***: This is the single-bit output that signals if the byte in the data_out output has been encrypted or decrypted.
- ***valid_out***: This is the single_bit output that helps identify if the byte in the *data_out* output is valid or not.

4.2 Functionalities

The following are workflows for encryption and decryption functionalities.
You can alternate the two functions every clock cycle.

4.2.1 Encryption

Workflow for encryption functionality:

1. Reset the machine using the asynchronous active low *rst_n* signal.
2. For the first byte of a new plaintext:
   - Set *data_in* to the first byte of plaintext.
   - Set *key* to the value of the key.
   - Set *new_message* to 1.
   - Set valid_in to 1.
   - Set *encrypt_in* to 1.
3. For every new byte of the same plaintext:
   - Set *data_in* to the current byte of plaintext.
   - Set *new_message* to 0.
   - Set *valid_in* to 1.
   - Set *encrypt_in* to 1.
4. Repeat from point 2 for every new plaintext.
5. If no new messages are given and all bytes inside the pipeline have been processed, the machine will wait for new inputs.

4.2.2 Decryption

Workflow for Decryption functionality:

1. Reset the machine using the asynchronous active low *rst_n* signal.
2. For the first byte of a new ciphertext:
   - Set *data_in* to the first byte of ciphertext.
   - Set *key* to the value of the key.
   - Set *new_message* to 1.
   - Set *valid_in* to 1.

- Set *encrypt_in* to 0.

3. For every new byte of the same ciphertext:
   - Set *data_in* to the current byte of ciphertext.
   - Set *new_message* to 0.
   - Set *valid_in* to 1.
   - Set *encrypt_in* to 0.
4. Repeat from point 2 for every new ciphertext.
5. If no new messages are given and all bytes inside the pipeline have been processed, the machine will wait for new inputs.

4.3 Expected Behavior

The machine will present the processed data on the output with a latency of 4 clock cycles, but it will produce a processed byte every clock cycle with a full pipeline. The design also uses an active-low asynchronous reset (*rst_n*) to guarantee immediate clearing of all pipeline registers regardless of the clock state. The machine can handle both plaintext and ciphertext bytes, provided the correct key and input flags, making it possible to implement in many operational conditions, including mixed encrypt/decrypt tasks, where it will reach 1 byte per clock cycle anyway.

The Figure 4.1 shows the pipeline's behavior when a new message is presented on input, following the previously described steps.

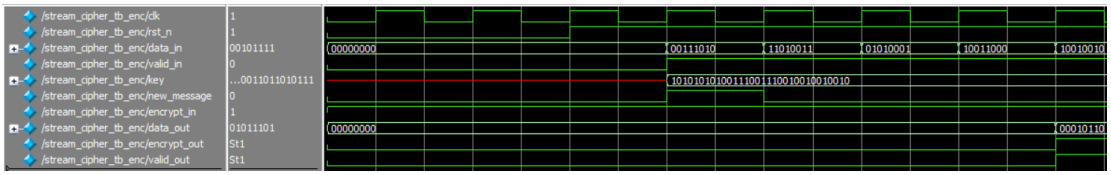

Figure 4.1: *Initialization of a new message.*

In the Figure 4.2, the ability to produce one byte for each clock cycle is shown.

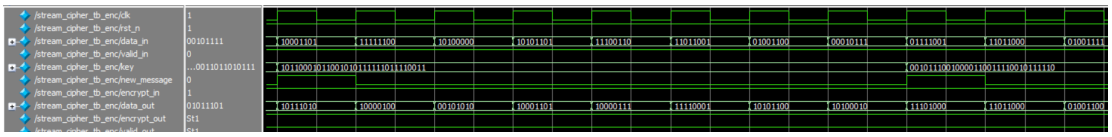

Figure 4.2: *One byte per clock cycle.*

In the end, Figure 4.3 shows the finalization of the pipeline operations.

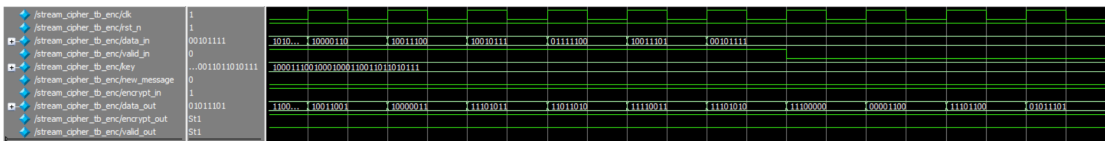

Figure 4.3: *Finalization of the pipeline operations.*

CHAPTER 5

Functional Verification

This chapter describes how the design was tested to verify its functionality according to the expected behavior described in Chapter 4.

5.1 Integrity tests

Tests *s_tb* and *cb_tb* aim to verify the basic functionality of the implementation, including the correctness of the counter block production and the *S()* function implementation. This helps during the development phase to ensure each step of the pipeline is correct.

5.1.1 s_tb

The *s_tb* module initializes the *S* module, feeds the data_in input of the *S* module with the value obtained from the counter block implementation in the gold model, and then compares the module's output with the expected outputs. This test helps identify implementation errors during development. If no errors are encountered, it will print only the string "test completed" as shown in Figure 5.1.

5.1.2 cb_tb

Similar to the *s_tb* test, the *cb_tb* test initializes the *counter_block* module and tests it using test values obtained by the golden model. The expected values are the output for the counter block implementation in the high-level model. As in the previous test, it will print only the string test completed if no errors are encountered, as shown in Figure 5.2.

```
VSIM 4> vsim s_tb
# End time: 17:02:55 on Nov 16,2025, Elapsed time: 0:04:03
# Errors: 0, Warnings: 2
# vsim s_tb
# Start time: 17:02:55 on Nov 16,2025
# Loading sv_std.std
# Loading work.s_tb
# Loading work.s
VSIM 5> add wave *
VSIM 6> run -all
# Test completed
```

Figure 5.1: *Output response of s_tb test.*

```
ModelSim> vsim cb_tb
# vsim cb_tb
# Start time: 17:03:31 on Nov 16,2025
# Loading sv_std.std
# Loading work.cb_tb
# Loading work.counter_block
VSIM 9> add wave *
VSIM 10> run -all
# Test completed
```

Figure 5.2: *Output response of cb_tb test.*

5.2 stream_cipher_tb_enc

This module implements a test for the design's encryption operation. The way to do this is the following: Two processes run in parallel: one implements the top-level stream cipher that produces the data stream, and the other captures the output and compares it with the test vectors generated by the golden model. This helps synchronize the two modules to avoid misalignment between the output data and captured data.

The first process runs the stream cipher module and charges the test vectors from memory. Then it sends each plaintext byte from the test vectors to the device under test (DUT) and sets the input signals appropriately.

```
initial begin: pipeline
    reg [63:0] plaintext [0:9];
    reg [31:0] keys [0:9];

    $readmemh("../modelsim/tv/plaintexts.mem", plaintext);
    $readmemh("../modelsim/tv/keys.mem", keys);

    $display("==== INIZIO TEST BYTE SINGOLI ====");

    clk = 0;
    rst_n = 0;
    data_in = 0;
    valid_in = 0;
    new_message = 0;

    repeat (3) @(posedge clk);
    rst_n = 1;
    @(posedge clk);
    encrypt_in = 1'b1;
    for (int j = 0; j < 10; j++) begin
        key = keys[j];
        new_message = 1'b1;
        for (int i = 0; i < 8; i++) begin
            data_in = plaintext[j][(7 - i)*8 +: 8];
            valid_in = 1'b1;
            @(posedge clk);
            new_message = 1'b0;
            valid_in = 1'b0;
        end
    end

end
```

Figure 5.3: *Snippet of code for the data processor of the encryption test*

The second one instead checks whether every DUT's output is correct, printing the result of the check by comparing the expected ciphertext with the output ciphertext, verifying if the encrypt_in signal is high for the encryption operation. The results output is shown in Figure 5.5, where only the last two tests are reported for brevity.

```
initial begin: data_capturer
    reg [63:0] expected_ciphertext [0:9];
    integer i = 0;
    integer j = 0;
    $readmemh("../modelsim/tv/ciphers.mem", expected_ciphertext);
    repeat(2) @(posedge clk);
    repeat (LATENCY + 1) @(posedge clk);
    forever begin
        @(posedge clk);
        if (valid_out) begin
            if (data_out == expected_ciphertext[j][(7 - i)*8 +: 8] && encrypt_out==1'b1) begin
                $display("Test %0d:%0d Passed: got %h, expected %h, encription: %h", j, i, data_out, expected_ciphertext[j][(7 - i)*8 +: 8], encrypt_out);
            end else begin
                $display("Test %0d:%0d Failed: got %h, expected %h", j, i, data_out, expected_ciphertext[j][(7 - i)*8 +: 8]);
            end
            i = i + 1;
            if (i == 8) begin
                i = 0;
                j = j + 1;
            end
            if (j == 10) begin
                $display("==== FINE TEST ====");
                $finish;
            end
        end
    end
end
```

Figure 5.4: *Snippet of code for the data capturer of the encryption test*

```
Test 8:0 Passed: got a7, expected a7, encription: 1
Test 8:1 Passed: got bd, expected bd, encription: 1
Test 8:2 Passed: got e6, expected e6, encription: 1
Test 8:3 Passed: got 1b, expected 1b, encription: 1
Test 8:4 Passed: got 43, expected 43, encription: 1
Test 8:5 Passed: got cf, expected cf, encription: 1
Test 8:6 Passed: got 99, expected 99, encription: 1
Test 8:7 Passed: got 83, expected 83, encription: 1
Test 9:0 Passed: got eb, expected eb, encription: 1
Test 9:1 Passed: got da, expected da, encription: 1
Test 9:2 Passed: got f3, expected f3, encription: 1
Test 9:3 Passed: got ea, expected ea, encription: 1
Test 9:4 Passed: got e0, expected e0, encription: 1
Test 9:5 Passed: got 0c, expected 0c, encription: 1
Test 9:6 Passed: got ec, expected ec, encription: 1
Test 9:7 Passed: got 5d, expected 5d, encription: 1
==== FINE TEST ====
```

Figure 5.5: *Results of the encryption test*

5.3 stream_cipher_tb_dec

This module implements a test for the design's decryption operation. It operates as the *stream_cipher_tb_enc* module, but provides as input every byte of the ciphertext of the test vectors and compares them with the plaintexts. This is also done to verify the correctness of the decryption operation.

```
initial begin: pipeline
    reg [63:0] ciphertext [0:9];
    reg [31:0] keys [0:9];

    $readmemh("../modelsim/tv/ciphers.mem", ciphertext);
    $readmemh("../modelsim/tv/keys.mem", keys);

    $display("==== INIZIO TEST BYTE SINGOLI ====");

    clk = 0;
    rst_n = 0;
    data_in = 0;
    valid_in = 0;
    new_message = 0;

    repeat (3) @(posedge clk);
    rst_n = 1;
    @(posedge clk);
    encrypt_in = 1'b0;
    for (int j = 0; j < 10; j++) begin
        key = keys[j];
        new_message = 1'b1;
        for (int i = 0; i < 8; i++) begin
            data_in = ciphertext[j][(7 - i)*8 +: 8];
            valid_in = 1'b1;
            @(posedge clk);
            new_message = 1'b0;
            valid_in = 1'b0;
        end
    end

end
```

Figure 5.6: *Snippet of code for the data processor of the decryption test*

```
initial begin: data_capturer
    reg [63:0] expected_plaintext [0:9];
    integer i = 0;
    integer j = 0;
    $readmemh("../modelsim/tv/plaintexts.mem", expected_plaintext);
    repeat(2) @(posedge clk);
    repeat (LATENCY + 1) @(posedge clk);
    forever begin
        @(posedge clk);
        if (valid_out) begin
            if (data_out == expected_plaintext[j][(7 - i)*8 +: 8] && encrypt_out==1'b0) begin
                $display("Test %0d:%0d Passed: got %h, expected %h, encryption: %h", j, i, data_out, expected_plaintext[j][(7 - i)*8 +: 8], encrypt_out);
            end else begin
                $display("Test %0d:%0d Failed: got %h, expected %h, encription: %h", j, i, data_out, expected_plaintext[j][(7 - i)*8 +: 8], encrypt_out);
            end
            i = i + 1;
            if (i == 8) begin
                i = 0;
                j = j + 1;
            end
            if (j == 10) begin
                $display("==== FINE TEST ====");
                $finish;
            end
        end
    end
end
```

Figure 5.7: *Snippet of code for the data capturer of the decryption test*

```
Test 8:0 Passed: got 08, expected 08, encryption: 0
Test 8:1 Passed: got 13, expected 13, encryption: 0
Test 8:2 Passed: got 4b, expected 4b, encryption: 0
Test 8:3 Passed: got b7, expected b7, encryption: 0
Test 8:4 Passed: got e8, expected e8, encryption: 0
Test 8:5 Passed: got 65, expected 65, encryption: 0
Test 8:6 Passed: got 30, expected 30, encryption: 0
Test 8:7 Passed: got 2b, expected 2b, encryption: 0
Test 9:0 Passed: got 90, expected 90, encryption: 0
Test 9:1 Passed: got ae, expected ae, encryption: 0
Test 9:2 Passed: got 86, expected 86, encryption: 0
Test 9:3 Passed: got 9c, expected 9c, encryption: 0
Test 9:4 Passed: got 97, expected 97, encryption: 0
Test 9:5 Passed: got 7c, expected 7c, encryption: 0
Test 9:6 Passed: got 9d, expected 9d, encryption: 0
Test 9:7 Passed: got 2f, expected 2f, encryption: 0
==== FINE TEST ====
```

Figure 5.8: *Results of the encryption test*

5.4 stream_cipher_tb_dec_and_enc

The last test performed aims to verify whether the pipeline can handle ciphertext and plaintext processing simultaneously and whether the design can process data arriving at any clock cycle rather than sequentially.

To do this, the test sends two different messages, the plaintext and the corresponding ciphertext, separately from one clock cycle to the other. Figure 5.9 shows the waveform of the test. It is possible to see that the bytes were received at a clock cycle interval and that their encrypt_in signals correspond to the operations performed. The Figure 5.8 shows the results of the test.

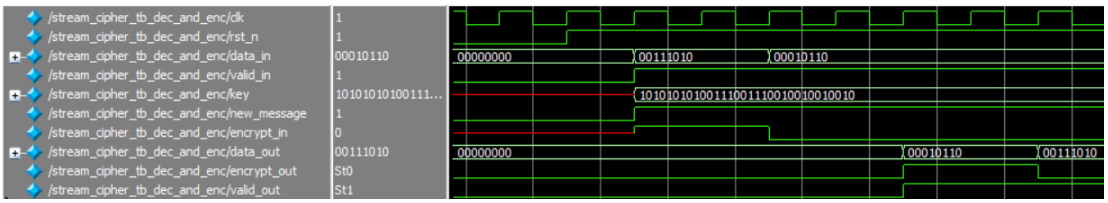

Figure 5.9: *Waveform of the stream_cipher_tb_dec_and_enc test*

```
# Test Passed: got 16, expected 16, encryption: 1
# Test Passed: got 3a, expected 3a, encryption: 0
```

Figure 5.10: *Results of the stream_cipher_tb_dec_and_enc test*

5.5 Conclusions on test operations

All functional tests passed successfully, confirming that the hardware implementation matches the golden model under all tested scenarios, including mixed encryption/decryption workloads and non-uniform data arrival patterns.

CHAPTER 6

FPGA Implementation Results

6.1 Synthesis Environment

The design was synthesized, placed, and routed using Quartus Prime 24.1 Lite, targeting an Intel Cyclone V 5CGXFC9D6F27C7 FPGA. All timing constraints were applied through an SDC file, including a single clock domain and appropriate I/O virtual constraints. The compilation included Analysis and Synthesis, Fitter, Assembler, and Timing Analysis.

6.2 Results and design refining

The first design, a 3-stage pipeline, met the 7.6 ns clock period requirement, but it was not satisfactory in terms of fMAX for the project's purpose. The timing analysis revealed a critical path between the S-function's output and the final output register. The combinational delay exceeded the desired clock period, so an additional pipeline stage was introduced after the XOR operation. This refinement reduced the critical path and significantly increased fMAX.

6.2.1 Timing analysis

The timing analysis was performed iteratively to ensure a fine result. After a few refinements, the analysis showed that the last design could handle the following constraints:

```
set CLK_PERIOD_NS 6.7
set MIN_IO_DELAY [expr double($CLK_PERIOD_NS)/10.0]
set MAX_IO_DELAY [expr double($CLK_PERIOD_NS)/5.0]
```

Figure 6.1: *Constraints in the clock signal*

This leads to an fMAX of 151.88 MHz in the Slow 1100 mV 85 °C corner.

6.2.2 Throughput

The pipeline produces one encrypted byte per clock cycle once filled. Therefore, the throughput is directly tied to the achievable clock frequency.

$$\text{Throughput} = f_{\text{MAX}} \times 8\,\text{bits} \tag{6.1}$$

So, with an fMAX of 151.88 MHz, the corresponding throughput will be around 1.21 Gb/s.

6.2.3 Latency

The pipeline consists of 4 stages, so the latency from *valid_in* to the corresponding *valid_out* is 4 clock cycles. This latency is constant and independent of the message length. Once the pipeline is filled, a new byte is produced every cycle.

6.2.4 Area Utilization

The post-fitting resource usage is extremely small compared to the device capacity, confirming that the stream cipher architecture is lightweight and highly resource-efficient. Table 6.1 summarizes the resource consumption.

Resource	**Used**	**Available**
Logic ALMs	58	113,560
Total registers	95	–
Block memory bits	0	12,492,800
RAM blocks	0	1,220
DSP blocks	0	342
PLLs	0	17
DLLs	0	4
I/O pins	1	378
Virtual pins	54	–

Table 6.1: *Post-fit resource utilization on Cyclone V.*

Overall, the area results demonstrate that the adopted architecture strikes a good balance between performance and hardware efficiency, achieving high throughput while keeping logic usage negligible.