UNIVERSITY OF PISA

MASTER'S DEGREE IN CYBERSECURITY

AES INVERSE BOX PROJECT #2 REPORT

COURSE HARDWARE AND EMBEDDED SECURITY

Author(s)
Riccardo Fantasia, Leonardo Pantani

Academic year 2023/2024

Contents

CHAPTER 1

Project Specifications

This project involved the design and implementation of an AES S-box based stream cipher that supports both encryption and decryption. The stream cipher utilises the AES Inverse S-box for transforming the 8-bit counter value during the encryption process. Unlike block ciphers, which process fixed-size blocks of data, stream ciphers operate on continuous streams of data, offering unique advantages in various applications.

The AES S-box-based stream cipher outlined in this project provides a secure and efficient method for encrypting and decrypting data. The AES Inverse S-box transformation and the properties of XOR operations are employed to ensure robust encryption while maintaining simplicity in design and implementation.

1.1 Encryption Algorithm

The core encryption mechanism of the stream cipher is based on XORing each byte of the plaintext with an 8-bit value derived from the AES inverse S-box transformation of an 8-bit counter value. The counter value, referred to as the counter block (CB), is initialized with an 8-bit symmetric key. The encryption can be mathematically represented as follows:

$$C[i] = P[i] \oplus S(CB[i])$$

where:

- $C[i]$ is the i-th byte of the ciphertext.
- $P[i]$ is the i-th byte of the plaintext.

- $CB[i]$ is the 8-bit value of the i-th counter block, calculated as $CB[i] = K + i \mod 256$.
- $S()$ denotes the Inverse S-box transformation from the AES algorithm.
- $\oplus$ is the XOR operator.

1.2 Decryption Algorithm

The decryption process in this stream cipher makes use of the properties inherent to the XOR operation. Given the encryption formula, the decryption is a relatively straightforward process, which is identical to the encryption process, with the exception of the replacement of P with C and vice versa.

$$P[i] = C[i] \oplus S(CB[i])$$

Given that XORing a value twice with the same key results in the original value being restored, it can be demonstrated that the decryption process successfully retrieves the original plaintext.

1.3 Design Specifications

The design of the stream cipher module must adhere to the following specifications:

- the module has an asynchronous active-low reset port called "reset_n" to ensure reliable initialization and reset functionality.
- the module has an input flag that indicates the validity and stability of the input data byte. It is called "valid_in" and is set to "1" when the data is stable, "0" otherwise.
- the module has an output flag that indicates the validity and stability of the output data byte. It is called "valid_out" and is set to "1" when the data is ready, "0" otherwise.
- for each new message integer, the counter block is reinitialized to the value of the 8-bit key.
- the module has a "new_message" flag that signals the beginning of a new message. It is set to "1" when a new message begins, "0" otherwise.

1.4 AES Inverse S-box Transformation

The Inverse S-box is implemented using a lookup table (LUT) in order to facilitate efficient development and faster computation.

1.4.1 Example

To illustrate, the application of the inverse S-box transformation to the byte 0x66 yields 0xD3 (row "6", column "6"), which is derived from the intersection of row 6 and column 6 in the inverse S-box table.

$$S(0x66) = 0xD3$$

CHAPTER 2

High-level Model

The high-level model was developed and tested in Python 3. The program accepts three optional arguments, which can be accessed via the -h/–help option.

- `-key` *(-k)* → the key that is used for encryption and decryption (in hexadecimal or decimal)
- `-input` *(-i)* → the values as integers (in hexadecimal or decimal) one after the other separated by a comma (no spaces)
- `-output` *(-o)* → if added, the script outputs three files that can be manually moved to the *modelsim/tv* folder for quick testing

In the event that the aforementioned parameters are not approved, the program will be executed with the default values that were also utilised to test the functionality of the hardware via the *Modelsim* software.

The script starts by defining the constant **SBOXTABLE** which contains an array containing all the values of the *S-BOX*.

The main function, `aes_stream_cipher`, receives 3 arguments:

- `key` → the key with which to encrypt and decrypt
- `data_array` → the array containing the values to be uniquely encrypted
- `encrypt` → boolean that holds if encrypted **TRUE**, **FALSE** otherwise

We initialise the `counter_block` as an array of length `data_array`, where each element is calculated as: `counter_block[i] = (key + i) % 256`. The use

of counter-blocks ensures that each byte of the message is encrypted with a unique value derived from the key provided as an argument.

Next, an empty `output_array` is created that will contain the results of the encryption or decryption. The function iterates over each element of the `data_array`, performing the following steps for each `i`:

- `data` ← `data_array[i]` ... *i-th* byte of the data array
- `cb_i` ← `counter_block[i]` ... *i-th* byte of the counter block
- `s_box_output` ← `SBOXTABLE[(cb_i >> 4) & 0xF][cb_i & 0xF]` ... finds in the S-Box using the first and the last 4 bits of `cb_i`
- `output_byte` ← `s_box_output` ⊕ `data` ... XOR between `s_box_output` and `data`
- `output_array.append(output_byte)` ... result gets added to the output array

At the end of the cycle, the function returns `output_array`.

CHAPTER 3

RTL Design

The first image below shows the diagram of the developed circuit. The upper part represents the graphic version of the `always_ff` code, while the one below represents the `always_comb`.

The second image represents the **Finite State Machine** showing the logic of the encryption and decryption circuit.

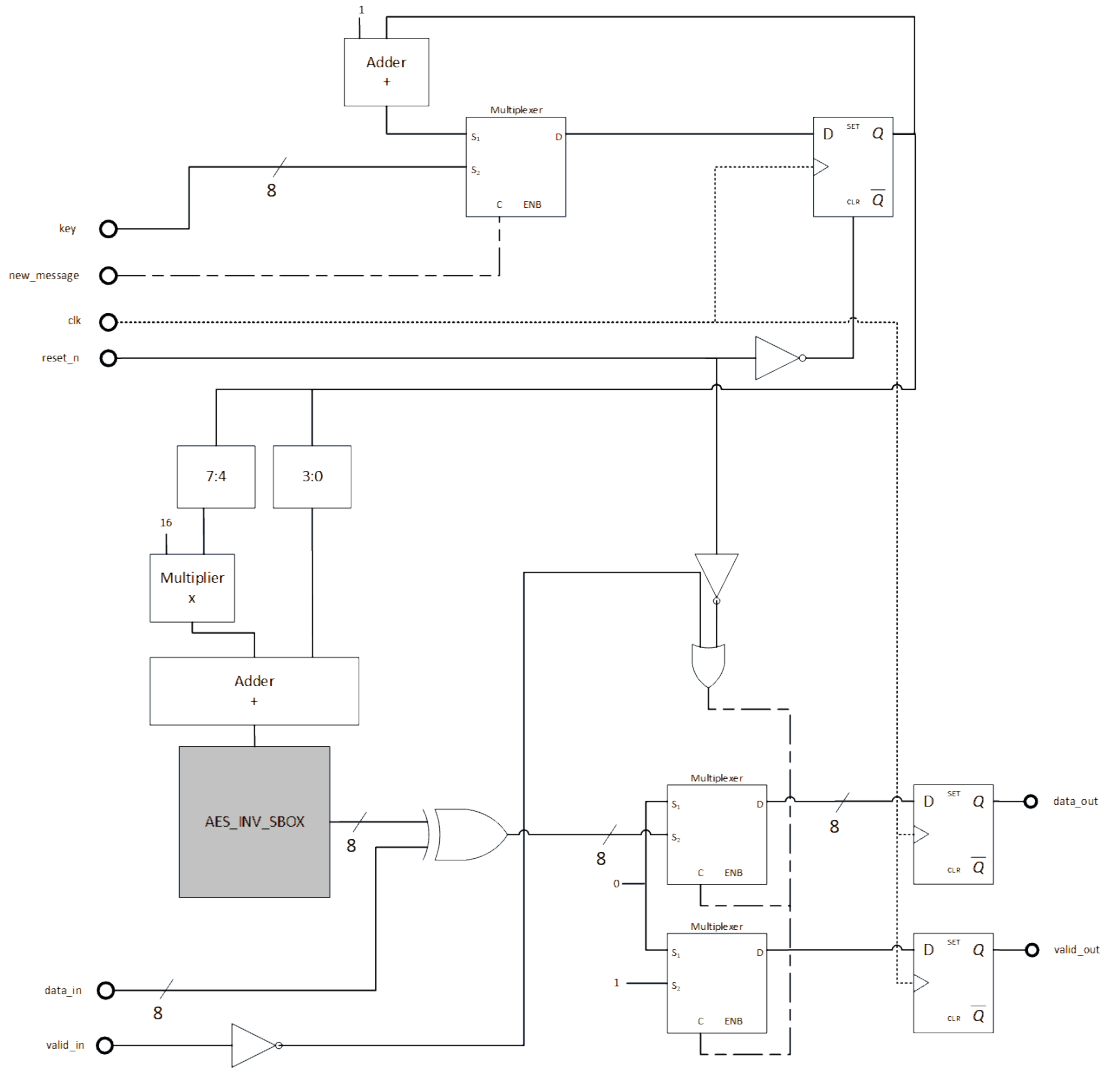

Figure 3.1: *Circuit diagram*

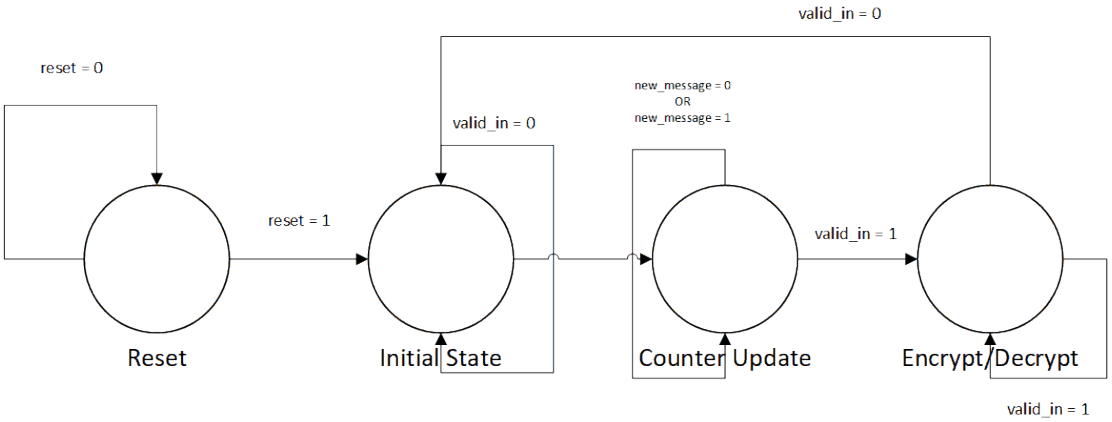

Figure 3.2: *Finite State Machine*

CHAPTER 4

Interface Specifications and Expected Behavior

The testbench is located at the path `tb/AES_cipher_testbench.sv`. This reads the files from the `modelsim/tv` folder and must be the following three:

- **input.txt** for each line the value to be encrypted must be specified in hexadecimal. It may have up to 256 values.
- **expected_output.txt** for each line the expected value must be specified in hexadecimal. It must have the same length as the input file.
- **key.txt** contains the key in hexadecimal with which to perform the encryption

To facilitate testing, use the *High-Level Model* explained in **Chapter 2** with the `-o` option in order to obtain test-bench-ready files.

CHAPTER 5

Functional Verification

5.1 Waveform

The testbench dynamically reads the number of lines and executes the test until the input finishes. The clock period is **20ps** and reset occurs after **15ps** from the start of the test.

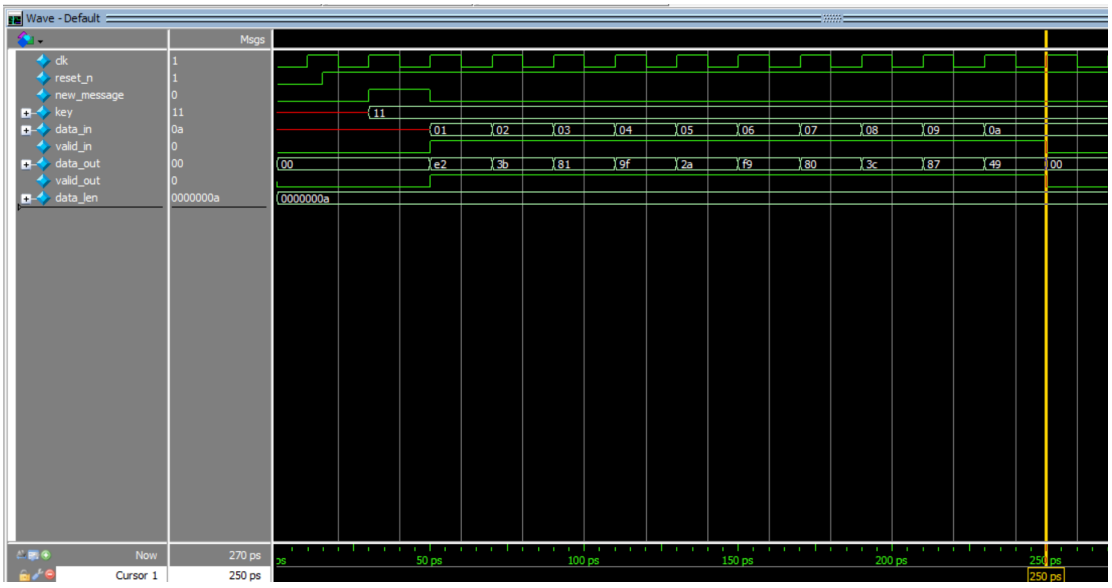

Figure 5.1: *Waveform obtained by running the high-level model with just the -o option*

5.2 Tests performed

We performed the tests using the high-level script with different input values, different keys and different input file lengths. The input file can be as long as 256 rows, one value per line. The testbench purposely terminates 2 clock cycles after the output of the for loop (which inserts the various lines of the test file into the loop) to verify that the signals `valid_out` and `data_out` are correctly reset to 0 to signal that the message to be encrypted has terminated. The only limiting case, if we want to consider it, is if the `new_message` value is set to **1** and no number to be encrypted is passed, in which case the output will be *invalid*; however, it has been considered that it is the user's responsibility to ensure that the signal is only set to **1** when there is a message to be encrypted and therefore we do not consider this behaviour to be incorrect.

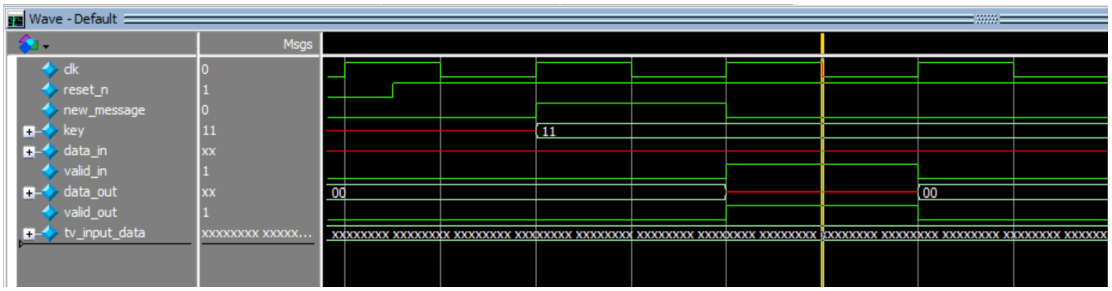

Figure 5.2: *Waveform obtained by running the simulation with an empty input file*

5.3 Test-bench composition

In the **AES_cipher_testbench** module, the necessary input and output signals for testing the AES circuit are declared. These signals include `clk`, `reset_n`, `new_message`, `key`, `data_in`, and `valid_in` for inputs, and `data_out` and `valid_out` for outputs. Initialization of a clock (clk) and a reset signal (reset_n) is carried out, along with the definition of `new_message` to indicate the availability of a new message and `valid_in` to signify the validity of the incoming data.

Furthermore, internal signals (the ones starting with *tv* or *internal*) are declared to facilitate loading test data from files.

The subsequent "Stimuli routine" initial block generates input signals for the AES circuit. The key file ("tv/key.txt") is read, and test input data is loaded ("tv/input.txt"). Subsequently, the testbench sends data to the AES circuit via the appropriate input signals, simulating the arrival of new messages to be encrypted.

In the "Check routine" initial block, the testbench verifies the output of the AES circuit. By waiting for the "valid_out" signal to be active, the testbench compares the circuit output with expected values ("tv_expected_output.txt"). In case of any discrepancies, an error message is displayed.

Finally, the testbench concludes the simulation using the "$stop" command.

CHAPTER 6

FPGA Implementation Results

To implement the circuit on the FPGA, the `AES_cipher.sv` file was first imported into the project. Then we added the file `time_constr_template.sdc` containing the **timing constraints**. Within the latter file, we specified in **line 12** a *false_path* from `reset_n` to `clk`. Finally, by trial and error, we modified the `CLK_PERIOD_NS` until we found a value that did not generate errors on *Quartus*, i.e. **10ns**.

We then started the compilation design operation, to perform both **Analysis and Synthesis**, **Fitter (Place & Route)**, **Assembler** and **Timing Analysis** simultaneously. It was verified that there were no *warnings* that could indicate an error during the design or configuration of the constraints. At the end of the **Timing Analysis**, the following results were obtained:

	Slow 1100mV 85C	**Slow 1100mV 0C**
FMax	102.12 MHz	101.5 MHz
Setup slack	0.208	0.104
Hold slack	0.741	0.441

Table 6.1: *Obtained results*