Politecnico di Milano – Department of Electronics, Information and Bioengineering

Optimized hardware implementation of the AES Cryptosystem

[Coding Project]

Student	Davide Sampietro
ID	87XXXX
Course	Embedded Systems
Academic Year	2018-2019
Advisor	Davide Zoni
Professor	William Fornaciari

September 10, 2019

Contents

1 Introduction

The aim of this project is providing an optimized hardware implementation for the **AES Cryptosystem** targeting an FPGA device.

We design the hardware accelerator using **System Verilog** as an HDL. We use **Vivado 2018.2** for the development, test and implementation of our project. The target platform is a *Xilinx Artix-7* FPGA (part no. XC7A100TCSG324C-1) mounted on a *Digilent Nexys4 DDR rev. C* board.

Since performance is the main driver of the project, we adopt an alternate implementation of the AES cryptosystem based on *Tboxes*. We squashed the trasformations included in the original round structure, `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` in a simpler one by mapping the AES *state bytes* through Tboxes. Tboxes are lookup tables that store precomputed values for the `SubBytes` and `MixColumns` operations combined together. Through fast Tbox *lookups* and simple *XOR* operations, it is possible to complete a whole round of the cipher in a single clock cycle.

Another design criterion is providing support for all of the three different AES key sizes, namely *AES-128*, *AES-192* and *AES-256*. Since the purpose of the project is producing an optimized AES core, we implement only the *ECB – Electronic Codebook* mode, which encrypts and decrypts each AES block separately.

The design is connected to the external world using a *UART* interface. The main features of the project are:

1. support for *AES-128*, *AES-192*, *AES-256* in *ECB* mode
2. Tbox-based implementation, completing a whole round in 1 clock cycle and a whole encyption/decryption in 15 clock cycles, irrespectively of the key size
3. unified encryption/decryption datapath, making the provided core easy to replicate in the same design
4. testing of the implementation on the final platform

2 AES Overview

AES is a *Symmetric Block Cipher* operating on **128-bit data blocks** and supporting three different key sizes (128/192/256-bit) according to the desired security level.

As many block ciphers, it processes its internal *state* (Figure 1), initialized either with the *plaintext* or *ciphertext*, in several rounds. They all share the same structure safe for the last (first) one during encryption (decryption).

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

Figure 1: Logical depiction of the AES internal state

The round is structured as follows:

1. `SubBytes` implements the non-linear layer (confusion) of the cipher. It operates a 8-bit to 8-bit transformation (usually implemented via simple lookups) based on GF(2^8) inverse calculation
2. `ShiftRows` performs the first step of the linear diffusion layer just by rotating the bytes on the same row of the state
3. `MixColumns` performs the second step of the linear diffusion layer by performing GF(2^8) multiplications by constants
4. `AddRoundKey` is a simple XOR with the round-key, and it's the secret dependent phase of the round

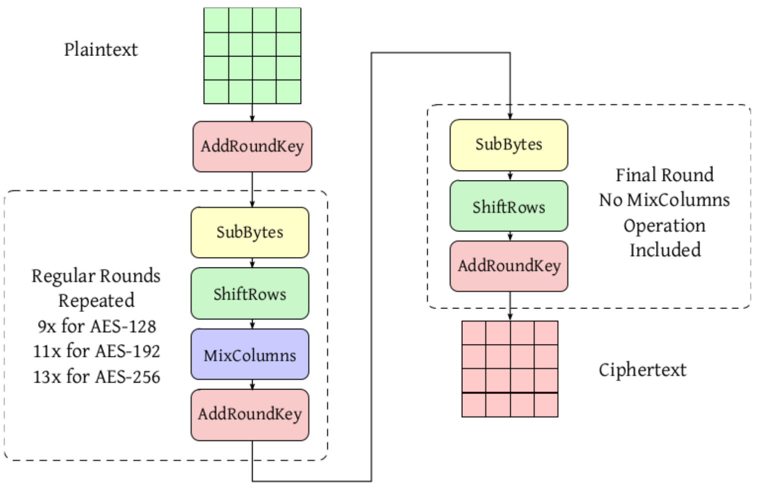

Figure 2: AES encryption operation

The overall encryption flow is depicted in Figure 2.

2.1 From Sboxes to Tboxes

The main intuition behind the Tbox-based implementation is that some of the round transformations of the AES cipher commute.

In particular, `SubBytes` and `ShiftRows` can be swapped immediately. This brings `SubBytes` and `MixColumns` close together. Tboxes are basically the result of merging these two phases. Instead of mapping state-bytes to Sboxes and then transforming them through `MixColumns`, we precompute the results and store them in a ROM for later use.

Each state-byte has to be looked-up in both of the approaches. However, while an Sbox maps it to an 8-bit value, a Tbox outputs a 32-bit word. At the cost of 4x ROM space, we can skip all the circuitry and latency associated to the `MixColumns` stage.

`MixColumns` operates on a 32-bit *state column* seen as a 4-byte column vector $\begin{bmatrix} s_0, & s_1, & s_2, & s_3 \end{bmatrix}$. The resulting transformation is:

$$\begin{bmatrix} s'_0 \\ s'_1 \\ s'_2 \\ s'_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \begin{bmatrix} s_0 \end{bmatrix} \oplus \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \begin{bmatrix} s_1 \end{bmatrix} \oplus \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \begin{bmatrix} s_2 \end{bmatrix} \oplus \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \begin{bmatrix} s_3 \end{bmatrix} \quad (1)$$

The multiplication of each constant column vector against a state byte defines a Tbox, which can be thus precalculated for each of the 256 values of the initial Sbox. It is not necessary to store the 4 Tboxes separately, as each one of them can be computed via a simple byte rotation. In particular, the original Tbox directly maps the state-bytes of the first row of the AES state. The Tboxes for the second/third/fourth row are obtained by rotating the original Tbox 1/2/3 bytes on the right.

Since `ShiftRows` is a simple rotation of the state-bytes on the same row of the AES state – which implies a trivial rewiring in hardware – the round structure of the AES cipher can be squashed in a single operation.

Let the AES state be the one in Figure 1. $s^k_{i,j}$ is a single state byte at the *k-th* round of the cipher. Let $TBOX(s,i)$ be a function that maps an 8-bit value s to a 32-bit word that is rotated i bytes to the right. Finally, let S^k_i with $i = 0,1,2,3$ be the *i-th* column of the 128-bit state at the *k-th* round and R^k_i the *i-th* column of the *k-th* round-key. Then:

$$\begin{aligned} S^k_i \quad = \quad TBOX(s^{k-1}_{0,i},0) \quad \oplus \quad TBOX(s^{k-1}_{1,i+1 \bmod 4},1) \quad \oplus \\ TBOX(s^{k-1}_{2,i+2 \bmod 4},2) \quad \oplus \quad TBOX(s^{k-1}_{3,i+3 \bmod 4},3) \quad \oplus \quad R^k_i \end{aligned} \quad (2)$$

Since we can perform all the Tbox lookups parallel given enough (i.e. 16) ROMs, and that the operations are simple XORs, this version of the AES cipher is particularly suitable for a fast hardware implementation. The final round, that skips the `MixColumns` transformation, just uses the output of the original Sboxes.

2.2 Inverse Cipher with Tboxes

The sequence of operations of the AES decryption round is the inverse of the encryption: `AddRoundKey`, `InvMixColumns`, `InvShiftRows`, `InvSubBytes`. While `InvShiftRows` and `InvSubBytes` still commute, the latter is a non-linear operation and thus doesn't commute with the linear permutation layer, that is, `InvMixColumns`.

However, since `InvMixColumns` is linear, it can be swapped with `AddRoundKey` given that also the round key undergoes the `InvMixColumns` transformation:

$$InvMixColumns(S^k_i \quad \oplus \quad R^k_i) \quad = \quad InvMixColumns(S^k_i) \quad \oplus \quad InvMixColumns(R^k_i) \quad (3)$$

By employing an inverse key schedule in which all the round keys, but the first and the last, have gone through the `InvMixColumns`, it is possible to obtain the exact same execution flow of the encryption, safe for:

1. alternate round keys, that need to be stored and incur in a 2x space overhead

2. inverse Tboxes, that are crafted in the same exact way as the direct one, by multiplying the constant vector $\begin{bmatrix} 0E, & 09, & 0D, & 0B \end{bmatrix}$ and the `InvSubBytes` outputs

This solution, adopted in our AES core, allows to totally reuse the encryption datapath for decryption by just selecting the alternate round key and the inverse Tbox values instead of the direct ones.

3 Implementation

The core we design is called `aeses_core`. It encompasses the logic for performing encryption/decryption using Tboxes and the key scheduler, that is run once and keeps the key material (both direct and inverse) in a proper storage.

It natively supports *AES-128*, *AES-192* and *AES-256*. To simplify the design, both the scheduling and encryption/decryption operations are padded to the length of the longest version of the cipher, *AES-256*. Thus, the encryption/decryption of a block requires 15 clock cycles irrespectively of the key size.

3.1 Key Scheduler

There is only one key scheduler, whose internal behaviour is also driven by the size of the input key.

The basic building block of the key scheduling is the g transformation, that takes as input a 32-bit word. It performs:

1. `SubBytes` transformation to each byte of the 32-bit word
2. 1-byte left rotation
3. XOR with a round constant, $RCON$

The bytes composing the key are arranged in columns just like the AES state depicted in Figure 1. There are 4 columns for a 128-bit key, 6 columns for a 192-bit key and 8 columns for a 256-bit key. The key scheduler outputs 11/13/15 128-bit round keys according to the key length. The way it crafts the several round keys slightly changes according to the key length.

Let W_i^k be the *i-th* column of the *k-th* key scheduling round.

1. Figure 3 shows the key scheduling round of AES-128. W_3^k goes through the g transformation, then is XORed with W_0^k to produce W_0^{k+1}. The remaining part of the state derives from a cascade of XORs between W_{i-1}^{k+1} and W_i^k with $i = 1, 2, 3$
2. Figure 4 shows the key scheduling round of AES-192. It is identical to the previous one, but this time the chain of XORs is longer in order to produce a 192-bit key scheduling state
3. Finally, Figure 5 shows the key scheduling round of AES-256. While the scheduler updates the first 128 bits of the state in the same exact way as AES-128, there is an additional `SubBytes` transformation involving W_3^{k+1} before going on with the chain of XORs.

In order to unify the datapath of the key scheduler for the three different key sizes, we keep a 256-bit state inside the key scheduler. It is split into 8 32-bit registers, named `key_word_register[0:7]`. When a new key schedule is requested, we first initialize them with the incoming 256-bit key – padded if necessary.

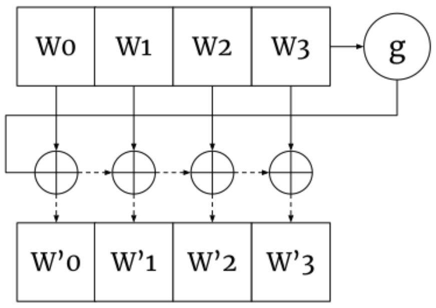

Figure 3: AES-128 key scheduling round

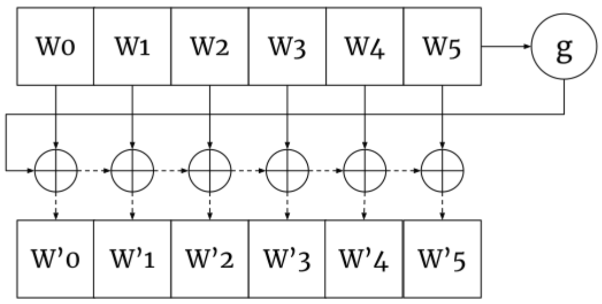

Figure 4: AES-192 key scheduling round

The key scheduler is capable of producing a 128-bit round key per clock cycle once it starts its operation. If the key size is 128-bit, we update only the lower `key_word_register[0:3]`, while the *write enable* of the other ones is disabled.

In a 256-bit schedule, we distinguish two phases: in the *even phases* we update, only the lower `key_word_register[0:3]`, while in the *odd phases*, only the upper `key_word_register[4:7]`.

In both these cases, when we produce the new internal state to be written at the end of the current clock cycle, we also output the old one already stored inside the registers.

Finally, in a 192-bit schedule, there are three phases. During the *first phase*, `key_word_register[0:5]` are updated. In the meanwhile, the old `key_word_register[0:3]` are output. The old `key_word_register[4:5]` are buffered in two additional registers, named `alternate_buffer[0:1]`. They would be otherwise overwritten during the *second phase*, when again `key_word_register[0:5]` are updated. The `alternate_buffer[0:1]` and the `key_word_register[0:1]` produced in the first phase are output. Finally, in the *third phase*, no state register is written and the last `key_word_register[2:5]` are output.

The correct output for each clock cycle is selected according to the phase number and the key size.

3.2 Encryption/Decryption Datapath

The structure of the encryption/decryption datapath is significantly easier. In fact, once we store the direct and inverse key schedule, the only thing that differentiates each key size is the number

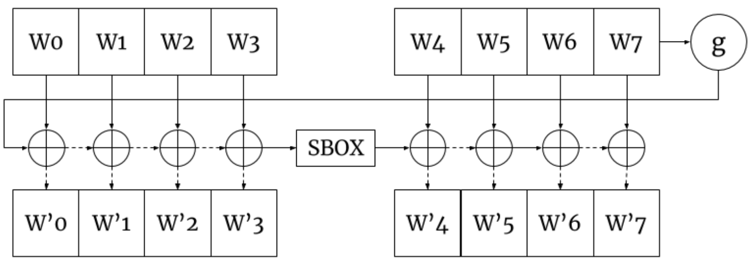

Figure 5: AES-256 key scheduling round

of rounds, which is 10/12/14 respectively for AES-128/192/256. This is achieved by initializing the `round_count` to 4/2/0 respectively, and ending the computation when it hits the value 13.

When `round_count` is equal to 13, no `MixColumns` or `InvMixColums` needs to be performed, thus the output consists of the outcome of the Sboxes rather than Tboxes, for both encryption and decryption.

Finally, thanks to the optimizations introduced in Section 2.2, we just need to either use direct or inverse Tboxes and key schedule to perform an encryption or decryption respectively. The inputs to this module, `round_key` and `inv_round_key` are supposed to be the correct ones since the moment the signal `start_operation` is asserted. The wrapping module has to ensure this invariant.

3.3 Host Controller

The `host_ctrl` module connects the provided `aeses_core` to a *UART* interface. The host controller is in charge of issuing commands to the AES core and collecting the results of encryptions and decryptions.

The original version only supported either encryption or decryption of AES-128. In order to extend the functionality of our implementation, the user now needs to send a so called `AESES_CONTROL_BYTE`, that instructs the `aeses_core` to: sit idle, create a new key schedule, encrypt or decrypt a 128-bit data block.

The format of the `AESES_CONTROL_BYTE` is defined in Table 1.

3.4 Main Components Interface

Here we present the input/output signals of the designed modules. All of them receive a `clk` and `rst` signal from the top level module, called `fpga_top` in the project sources. The reset is *synchronous* and *active high* throughout `aeses_core`.

3.4.1 `aeses_core`

The AES top-level module is shown in Figure 6.

It has three possible **internal states**:

1. `READY`: the core is ready to do something, be it a new key-schedule or and encryption/decryption;

Table 1: AESES_CONTROL_BYTE encoding

Bits	Function	Values	Behaviour
7	RESET New key schedule	0 – disabled 1 – enabled	Start a new key schedule and wait for 32 bytes from UART
6-5	Key size	00 – AES-128 01 – AES-192 1X – AES-256	Only taken into account if RESET=1
4	EN Enable AES	0 – idle 1 – enable core	Valid only if RESET=0. It waits for 16 data bytes from UART
3	ENC_DEC	1 – encryption 0 – decryption	Valid only if EN=1 and RESET=0
2-0	Unused	XXX	`AESES_CONTROL_BYTE` padding

2. `SCHEDULING`: the core is calculating a key-schedule;
3. `OPERATION`: the core is encrypting/decrypting a block of data.

A brief documentation of the **input signals**:

- `aes_blk_i`: input plaintext/ciphertext
- `aes_key_i`: input key for AES
- `enable_key_schedule_i`: when the core is `READY` and the signal is asserted for 1 clock cycle, start a new key scheduling operation with `aes_key_i`
- `enable_op_i`: when the core is `READY` and the signal is asserted for 1 clock cycle, start a new encryption/decryption operation. If both `enable_op_i` and `enable_key_schedule_i` are asserted, the latter has the precedence.
- `enc_decneg_i`: if 1, encrypt the data block, otherwise decrypt. It needs to be correctly set only when `enable_op_i` is asserted, it is otherwise ignored.
- `key_mode_i`: specifies the length of the key, according to the encoding established in Table 1. It needs to be correctly set only when `enable_key_sched_i` is asserted, it is otherwise ignored.

A brief documentation of the **output signals**:

- `ready_o`: asserted when the core is in the `READY` state
- `aes_blk_o`: this is the output of an AES encryption/decryption
- `valid_o`: asserted then `aes_blk_o` is valid

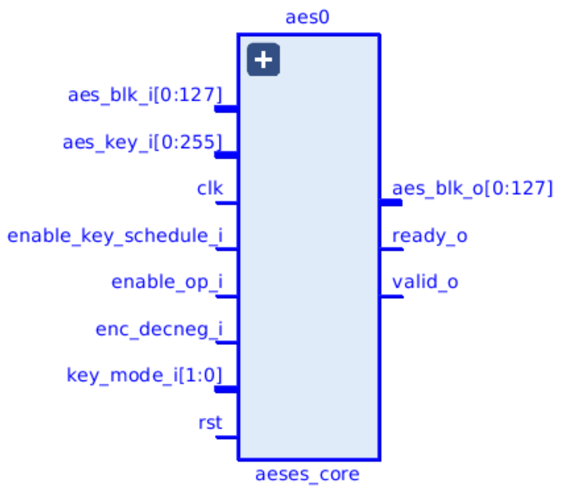

Figure 6: Interface of the `aeses_core`

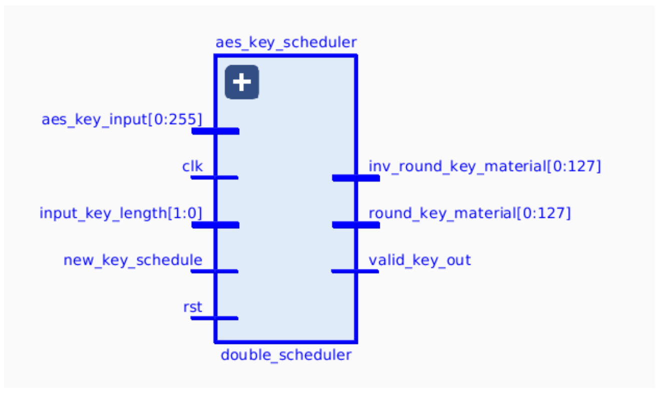

Figure 7: Interface of the `key_scheduler`

3.4.2 `double_scheduler`

The module performing the key scheduling operation is shown in Figure 7

A brief documentation of **input signals**:

- `aes_key_input`: the key being scheduled
- `input_key_lenght`: specifies the correct key length
- `new_key_schedule`: starts a new key scheduling operation

A brief documentation of **output signals**:

- `inv_round_key_material`: the round key for the inverse key schedule (decryption)
- `round_key_material`: the direct round key for encryption
- `valid_key_out`: set when the previous outputs are valid

The key-scheduler outputs round keys at the rate of 1 round key/clock cycle. The modules storing this information need to be fast enough to catch them, otherwise they are lost.

3.4.3 `shift_register`

It holds the round keys for encryption. It is shown in Figure 8.

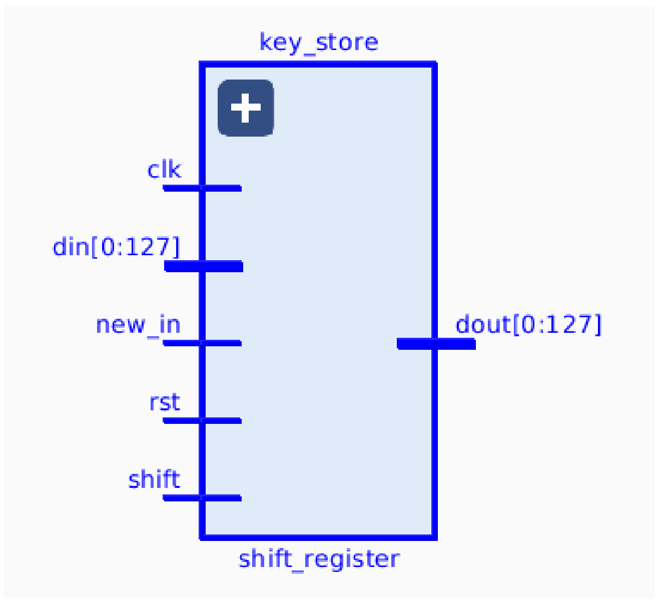

Figure 8: Interface of the `shift_register`

This module has a *read end* and a *write end* and does not allow random access to its internal storage.

During a write operation, the internal data-words are shifted forward in the next slot. If the size of the storage is exceeded, older data is lost.

During a read operation, the data from the read-end is output and the register is shifted forward. However, the output data is written back to the queue, thus implementing a circular data structure.

A brief documentation of **input signals**:

- `din`: input data
- `new_in`: writes a new chunk of data at the write-end
- `shift`: if set, move the shift register forward

A brief documentation of **output signals**:

- `dout`: data stored at the read end

The module `circular_lifo` behaves similarly, but it actually implements a "cirular" stack for storing the inverse key schedule.

3.4.4 `aes_cipher`

The module performing encyrption/decryption of AES blocks is shown in Figure 9.

A brief documentation of **input signals**:

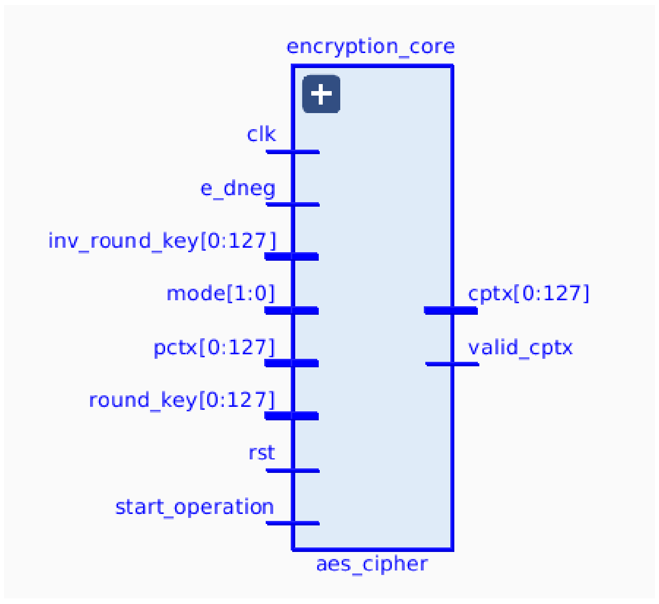

Figure 9: Interface of the `aes_cipher`

- `e_dneg`: needs to be correctly set during the whole encryption/decryption process. If 1, encrypt, otherwise decrypt.
- `start_operation`: when asserted for one clock cycle, start a new operation.
- `mode`: key schedule length. Needs to be correctly set only when `start_operation` is asserted.
- `pctx`: AES input block
- `round_key` and `inv_round_key`: the correct 128-bit words for the key addition phase.

A brief documentation of **output signals**:

- `cptx`: output AES block
- `valid_cptx`: if set for 1 clock cycle, the output is valid

Since the very moment `start_operation` is asserted, the module requires the correct round keys to materialize at its input ports. A round key is consumed every clock cycle, and if the top level module doesn't keep up with this rate, the results will be wrong.

3.5 `aeses_lite`

The lightweight implementation of *AES-256* for *ECB encryption* is based on a stripped down version of the full `aeses_core`. It has been designed to achieve the maximum possible working frequency and reduce the footprint.

4 Results

All the provided implementation complete an encryption/decryption operation in 15 clock cycles.
The results about peak-perfomance and utilization are shown in Table 2.

Table 2: Final results

	`aeses_core`	`aeses_lite`
UTILIZATION		
Slice LUT	12%	5%
F8 Muxes	6%	1%
F7 Muxes	6%	2%
Slice Register	3%	1%
PERFORMANCE		
Clock Cycles	15	15
Frequency (MHz)	166.67	227.27
Throughput (Gbit/s)	1.4	1.94

References

[1] NIST. Specification for the advanced encryption standard (aes). 2001.

[2] Barenghi A. Pelosi G. Appunti del corso cryptography and security of digital devices.

[3] Avi Kak. Lecture 8: Aes: The advanced encryption standard, 2018.