ELECH-409 – Digital Architectures and Design

VHDL project: AES encyption

HUO Jian

SALA MANGITUKA Glodi

MILOJEVIC Dragomir

VAN SLIPJE Oscar

ALI Muhammad

22-12-2023

Contents

1 General overview

1.1 Algorithm Explanation

The Advanced Encryption Standard is a symmetric encryption algorithm that operates on fixed-size blocks of data (128 bits). The algorithm consists of several steps performed in rounds to convert plaintext into ciphertext. The four main modules used in AES encryption are AddRoundKey, SubBytes, ShiftRows, and MixColumns. Here's a general overview of the algorithm and the description of each module.

- **Key Expansion**: The 128-bit encryption key undergoes a key expansion process to generate a key schedule. This schedule provides round keys for each round of encryption.
- **Initial Round**: The plaintext block is XORed with the initial round key.
- **Rounds**: The main encryption process consists of several rounds (typically 10 rounds for AES-128), each composed of the following steps:
  - **AddRoundKey**: XORs the state with the round key derived from the key schedule.
  - **SubBytes**: Substitutes each byte of the state with another byte from a fixed table (S-box).
  - **ShiftRows**: Performs a cyclic shift on the rows of the state matrix.
  - **MixColumns**: Mixes the columns of the state matrix using matrix multiplication.
- **Final Round**: The final round excludes the MixColumns step.

Each module will be individually explained and implemented to detail their specific operations in the AES encryption process, contributing to the overall encryption and ensuring the security of the data.

1.2 Block diagram

Hereafter you will find the block diagram of the final module generated by Vivado and two block diagrams we made of how our different modules are interconnected to each other, the first being simpler and the second being more detailed.

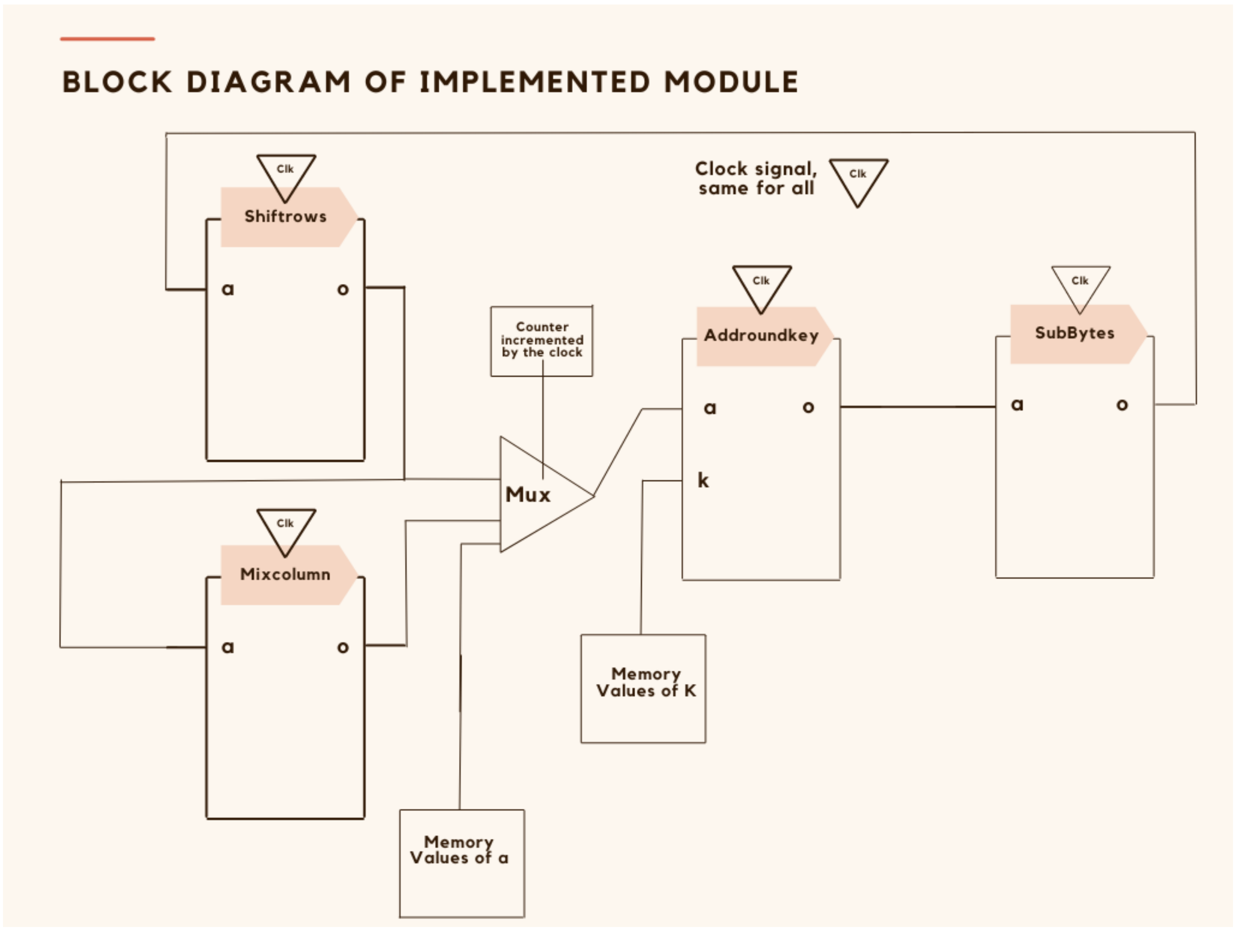


Figure 1.1: The simpler block diagram of how our different modules are interconnected to each other

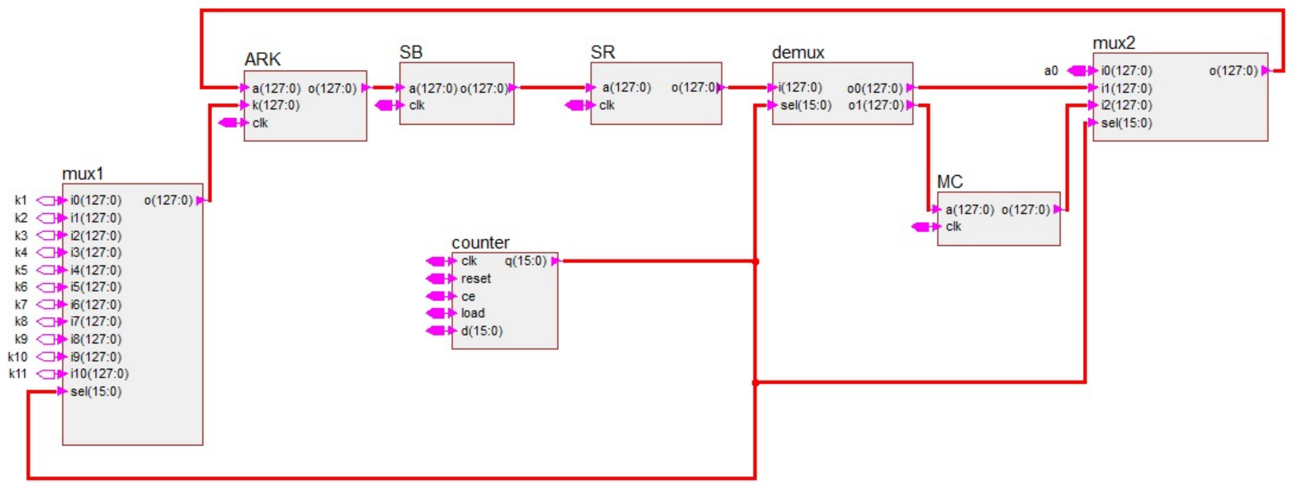


Figure 1.2: The more detailed block diagram of how our different modules are interconnected to each other

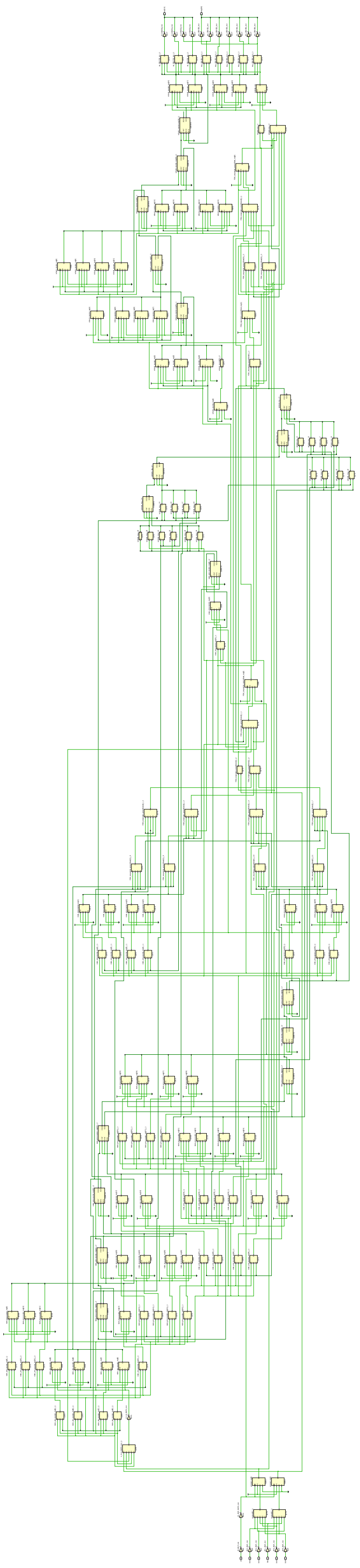


Figure 1.3: block diagram generated by vivado

2 Modules Implementations

2.1 AddRoundKey Module

2.1.1 Objectives

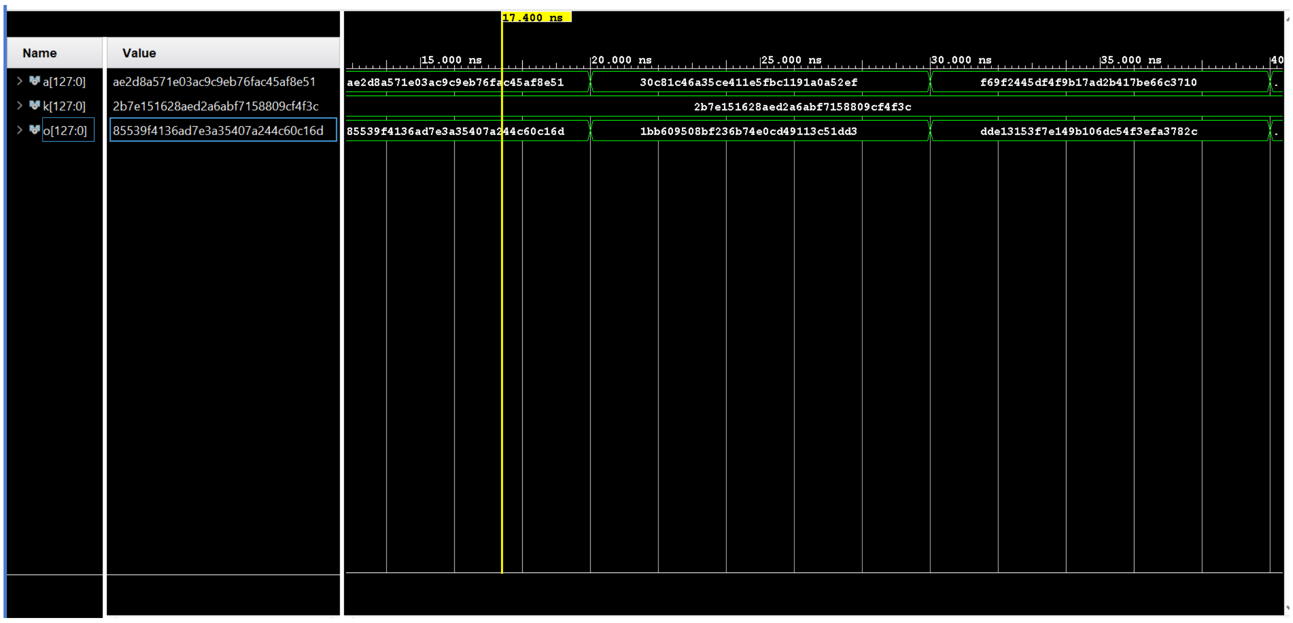
The 128-bit block, obtained from the previous step, undergoes the AddRoundKey operation where it is bitwise XORed with the specific RoundKey for that round. Each round uses a different RoundKey, as previously mentioned. These unique RoundKeys for each round are provided in the RoundKeys.txt file. The generation of these distinct RoundKeys is achieved through a key schedule procedure, although this key expansion procedure does not need to be implemented for this specific project's purposes.

2.1.2 Implementation

The *AddRoundKey* entity represents an XOR operation between two input vectors, a and k, both of which are 128-bit vectors. The entity's output o is also a 128-bit vector. Within the architecture section, the XOR operation is implemented simply as $o <= (a \oplus k)$, denoting that each bit of the a vector is XORed with the corresponding bit in the k vector to produce the output vector o.

2.1.3 Validation

In the test bench, the verification of the module's functionality is conducted using the provided examples. This involves assessing whether the inputs, represented by 'a' and a certain key, produce the expected output from the AddRoundKey operation. By comparing the computed output against the expected results, the test bench validates the correctness of the AddRoundKey module's implementation. In this case, we can confirm that the module is performing the XOR operation accurately and producing the expected output.



Name	Value
a[127:0]	ae2d8a571e03ac9c9eb76fac45af8e51
k[127:0]	2b7e151628aed2a6abf7158809cf4f3c
o[127:0]	85539f4136ad7e3a35407a244c60c16d

```
a: ae2d8a571e03ac9c9eb76fac45af8e51 | 30c81c46a35ce411e5fbc1191a0a52ef | f69f2445df4f9b17ad2b417be66c3710
k: 2b7e151628aed2a6abf7158809cf4f3c
o: 85539f4136ad7e3a35407a244c60c16d | 1bb609508bf236b74e0cd49113c51dd3 | dde13153f7e149b106dc54f3efa3782c
```

Figure 2.1: Waveform of the AddRoundKey module test-bench

2.2 SubBytes Module

2.2.1 Objectives

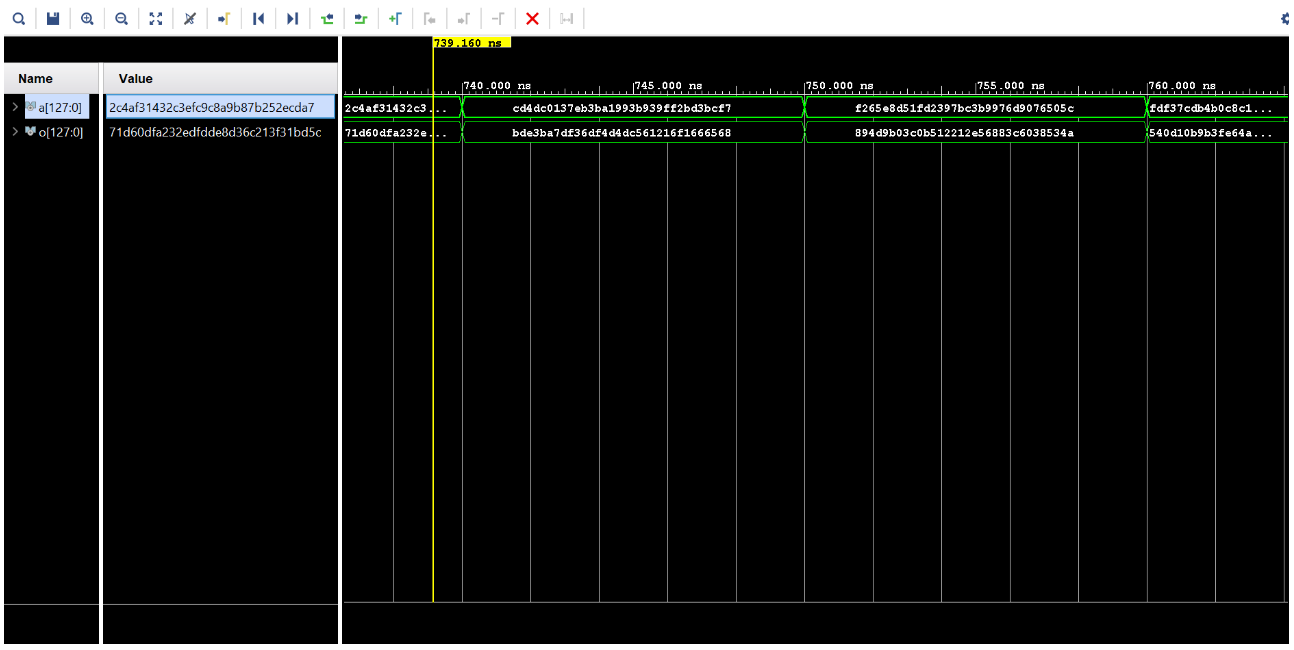
The SubBytes step involves substituting each byte utilizing an S-box, which serves as a substitution table. The S-box, available as a lookup table, is provided in the file named S box.vhd.

2.2.2 Implementation

The *SubBytes* module takes a 128-bit input vector *a* and produces a corresponding 128-bit output vector *o*. This implementation comprises a loop that iterates through the 128 bits in *a*, dividing it into 16 bytes (each 8 bits long). Within this loop, for each byte, the component '*S_box*' is instantiated to perform the byte-wise substitution. The '*S_box*' component takes an 8-bit input byte and provides an 8-bit output byte based on a predefined substitution table. The loop maps each byte of the input vector *a* to the '*S_box*' component, conducting byte-wise substitutions and assembling the resulting bytes into the output vector *o*.

2.2.3 Validation

In the test bench for this module, the verification process follows the same procedure as in the previous module. The test bench involves altering the input values and subsequently observing whether the output values align with the expected results. This procedure validates the functionality of the module by confirming that the output corresponds correctly to the modified input values.



```
Name        Value
a[127:0]    2c4af31432c3efc9c8a9b87b252ecda7
o[127:0]    71d60dfa232edfdde8d36c213f31bd5c
```

739.160 ns
740.000 ns
745.000 ns
750.000 ns
755.000 ns
760.000 ns
2c4af31432c3...
cd4dc0137eb3ba1993b939ff2bd3bcf7
f265e8d51fd2397bc3b9976d9076505c
fdf37cdb4b0c8c1...
71d60dfa232e...
bde3ba7df36df4d4dc561216f1666568
894d9b03c0b512212e56883c6038534a
540d10b9b3fe64a...

Figure 2.2: Waveform of the SubBytes module test-bench

2.3 ShiftRows Module

2.3.1 Objectives

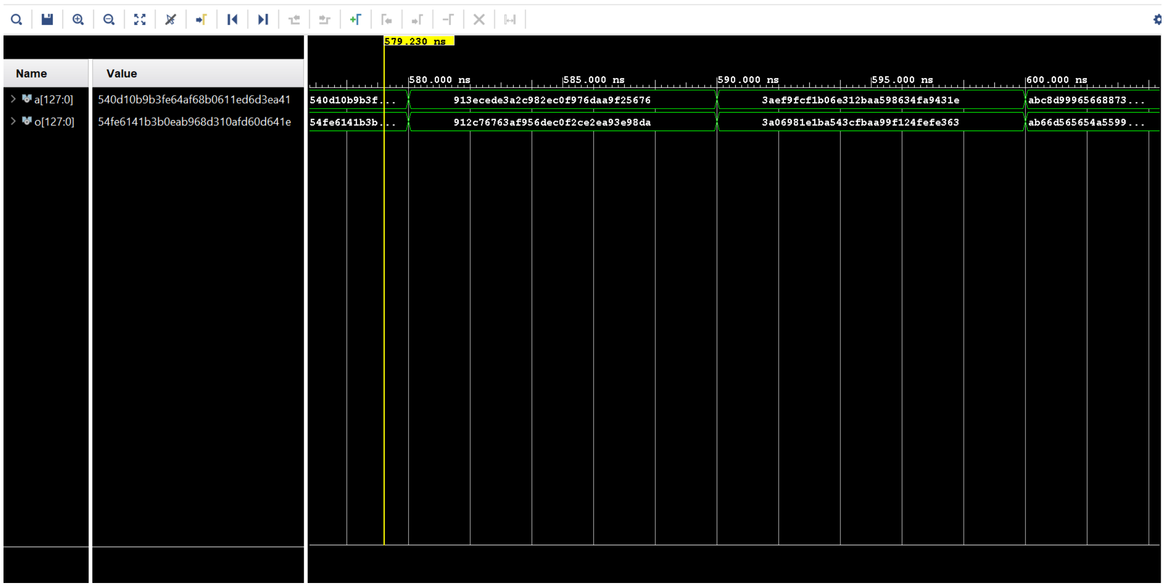
The 'shiftRows' module follows a specific pattern: it cyclically shifts the bytes in the last three rows to the left. The number of bytes shifted is equal to the respective row number. However, the first row remains unchanged and is not subjected to any shifting operation.

2.3.2 Implementation

For each element in the matrix representation of the input vector, the shifting is performed based on the row number. To move across columns, the process involves shifting by 32 bits (since the elements are 32 bits wide), while moving between rows requires a shift of 8 bits (as each row consists of 8-bit elements). When i equals zero (representing the first row), no shifting occurs. For rows 1, 2, and 3, the shifting logic is implemented by calculating the iterator variable to determine the correct shift distance for each element. The output o is constructed by rearranging the bits from the input a according to the shifting logic based on row and column indices, producing the resulting shifted rows.

2.3.3 Validation

Validation within the test bench for the 'ShiftRows' module involves modifying input values and checking if the output aligns with the expected results. It ensures the shifting operation manages cyclic left shifts within the last three rows accurately. Simultaneously, it upholds the integrity of the first row. That confirms the module's functionality, mirroring the earlier validation procedure.



```
Name       Value
a[127:0]   540d10b9b3fe64af68b0611ed6d3ea41
o[127:0]   54fe6141b3b0eab968d310afd60d641e

579.230 ns
580.000 ns   585.000 ns   590.000 ns   595.000 ns   600.000 ns
540d10b9b3f...  913ecede3a2c982ec0f976daa9f25676  3aef9fcf1b06e312baa598634fa9431e  abc8d99965668873...
54fe6141b3b...  912c76763af956dec0f2ce2ea93e98da  3a06981e1ba543cfbaa99f124fefe363  ab66d565654a5599...
```

Figure 2.3: Waveform of the ShiftRows module test-bench

2.4 MixColomns

2.4.1 Objectives

In AES, MixColumns multiplies each input column by a fixed matrix using Galois field arithmetic. The matrix used is:

$$c(x) = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

For instance, to compute $b_{0,0}$, multiply $c(x)$ with the input column:

$$\begin{bmatrix} b_{0,0} \\ b_{1,0} \\ b_{2,0} \\ b_{3,0} \end{bmatrix} = c(x) \cdot \begin{bmatrix} a_{0,0} \\ a_{1,0} \\ a_{2,0} \\ a_{3,0} \end{bmatrix}$$

Then, calculate $b_{0,0}$ as:

$$b_{0,0} = (2 \cdot a_{0,0}) \oplus (3 \cdot a_{1,0}) \oplus (a_{2,0}) \oplus (a_{3,0})$$

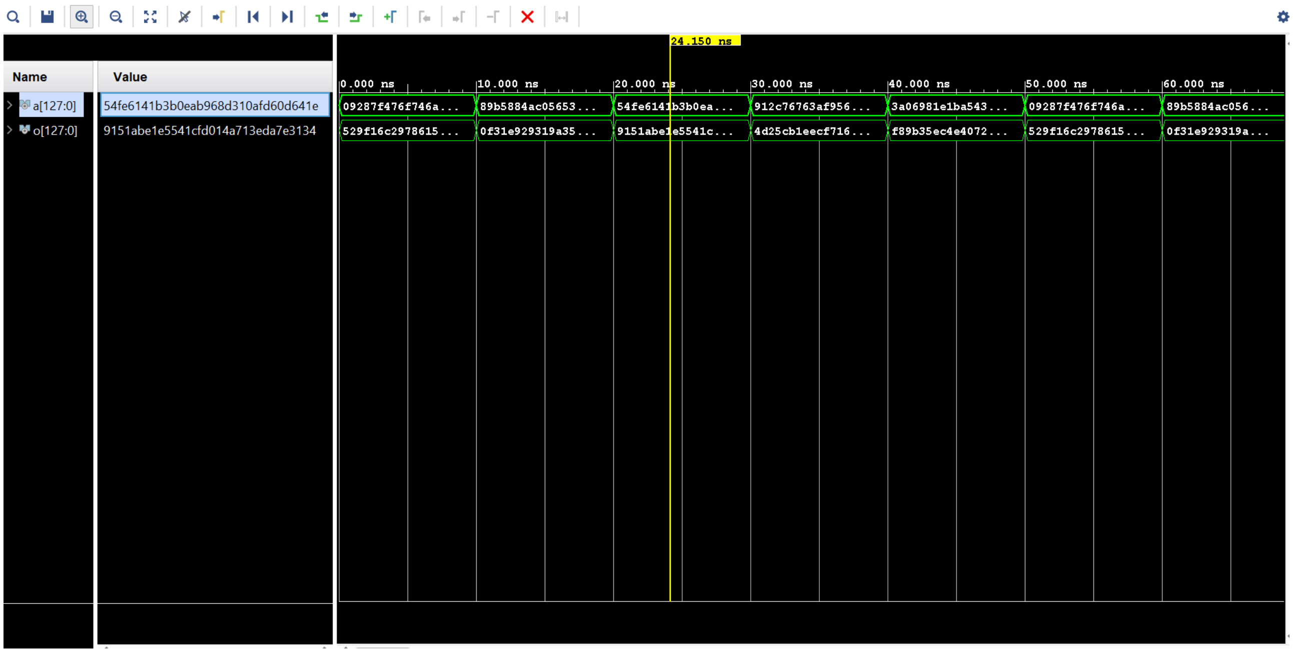
The values $(2 \cdot a_{0,0})$ and $(3 \cdot a_{1,0})$ are obtained using Look-Up Tables (LUTs) named 'mul2.vhd' and 'mul3.vhd' respectively. Similar computations are performed for other bytes in the column.

2.4.2 Implementation

The *MixColumn* module performs matrix multiplication on the input columns utilizing Galois field arithmetic. It is composed of Look-Up Tables 'LUT_mul2' and 'LUT_mul3', used for specific multiplications. The process involves iterating through the input columns and applying operations based on the row and column indices. For each element in the output vector o, the value is determined according to the specific matrix multiplication logic governed by the switch-case statement. The computation of each byte in the output vector involves XOR operations between appropriate elements from 'mul2', 'mul3', and the input a. The output o is constructed by assigning calculated values to specific bit ranges according to the iteration indices, resulting in the transformed columns after matrix multiplication and Galois field arithmetic operations on the input bytes.

2.4.3 Validation

The test bench for validating the MixColumn module employs a methodology akin to the previous validation processes. It inspects rigorously the output against expected vectors derived from predefined examples specific to the MixColumn step. We can thus confirming its accuracy in transforming the input columns.



```
Name      Value
a[127:0]  54fe6141b3b0eab968d310afd60d641e
o[127:0]  9151abe1e5541cfd014a713eda7e3134
```

Figure 2.4: Waveform of the Mixcolomn module test-bench

3 AES Encryption

3.1 AES encryption module

3.1.1 Objectives

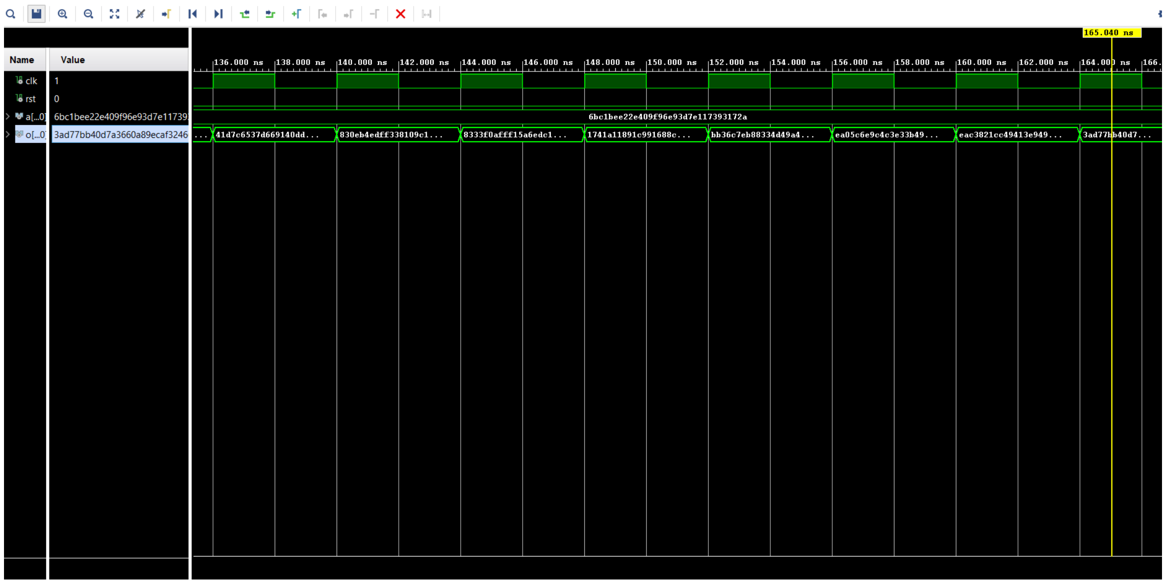
The primary aim at hand is to effectively utilize the four implemented modules in the specified sequence to achieve the encryption process. This entails orchestrating the AddRoundKey, SubBytes, ShiftRows, and MixColumns modules in the required order to execute the encryption algorithm accurately. The task revolves around integrating these modules systematically to ensure the smooth execution of the encryption process. This alignment enables the generation of encrypted data by efficiently coordinating the individual functionalities encapsulated within each module. It will culminate in the successful completion of the encryption protocol.

3.1.2 Implementation

The 'StepsModules' encapsulates the AES encryption module, consisting of interconnected submodules like AddRoundKey, SubBytes, ShiftRows, and MixColumn. Governed by a state machine controlled by 'currentState' and 'nextState' signals, the system progresses through different states ('SB', 'SR', 'MC', 'ARK', 'RSTT') to execute distinct steps of the AES encryption process.

The system begins at the 'RSTT' state upon receiving a reset signal ('rst = '1"), initializing the encryption process. The 'counter' variable manages the state transitions and controls the number of encryption rounds. The 'SubBytes', 'ShiftRows', 'MixColumns', and 'AddRoundKey' states correspond to specific AES encryption steps, sequentially performing byte substitution, row shifting, column mixing, and addition of the round key, respectively. These states progress in synchronization with the 'counter', which regulates the encryption rounds until the specified number of rounds (here, 10) is completed. Upon the completion of rounds, the system returns to the 'ARK' state to finalize the encryption process, producing the encrypted output 'o'. The 'clk' signal regulates the state transitions, ensuring the sequential execution of AES encryption steps. The 'rst' signal initiates the reset for the AES encryption process.

3.1.3 Validation



```
Name    Value
clk     1
rst     0
a[...0] 6bc1bee22e409f96e93d7e117393
o[...0] 3ad77bb40d7a3660a89ecaf3246
```

```
a: 6bc1bee22e409f96e93d7e117393172a
o: ... | 41d7c6537d669140dd... | 830eb4edff338109c1... | 8333f0afff15a6edc1... | 1741a11891c991688c... | bb36c7eb88334d49a4... | ea05c6e9c4c3e33b49... | eac3821cc49413e949... | 3ad77bb40d7...
```

Figure 3.1: Weveform of the AES encryption module test-bench

As depicted in the scheme in Figure 3.1, each step within the encryption protocol executes within a single clock cycle. The observed final value for the encryption of the first word indicates its correctness, validating the module's proper functionality.

3.2 Final module

3.2.1 Objectives

Having successfully implemented the encryption module utilizing the four components, the goal is now to deploy this module on the Basys3 board and enable encryption control using the board's buttons. The 'btnC' button will initiate the encryption process. Upon completion of encryption, the board's 7-segment display should show 'AES'. Additionally, 'btnR' will facilitate a restart of the encryption process and clear the 7-segment display. It's important to note that 'btnR' should only trigger the encryption restart and not initiate the process anew.

3.2.2 Verification

In Figure 3.2, it's evident that the encryption process initiates for the first time upon pressing 'btnC', and the encryption is performed on the initial word provided in the given list. Upon verification, the output of the encryption process matches the expected result, confirming its accuracy.

The zoomed-in waveform, in Figure 3.3, depicts the final module. Notably, the 'an' and 'seg' outputs, responsible for controlling the 7-segment display, indicate 'AES' upon completion of encryption. Observing the change in 'an' and 'seg' values confirms the accurate display of the intended text, and the output values

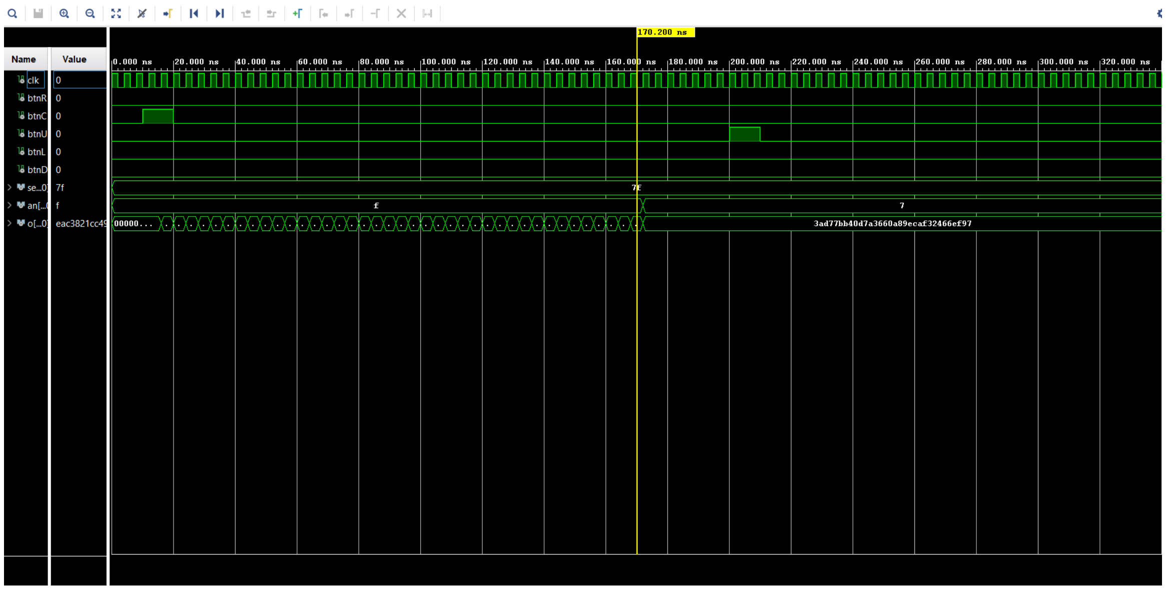


Figure 3.2: Waveform of the final module implemented part 1

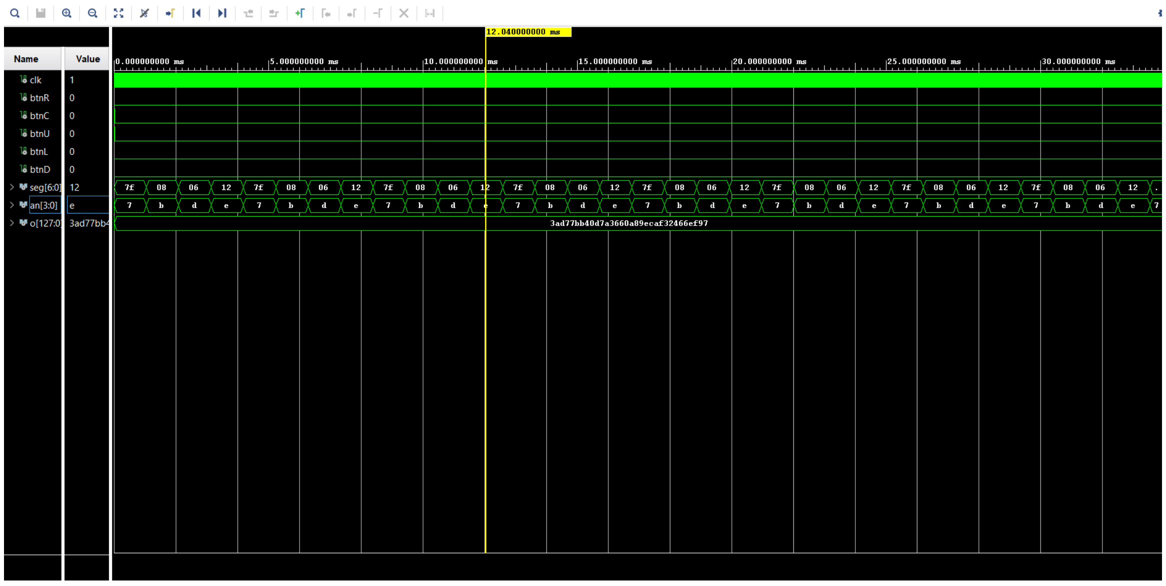
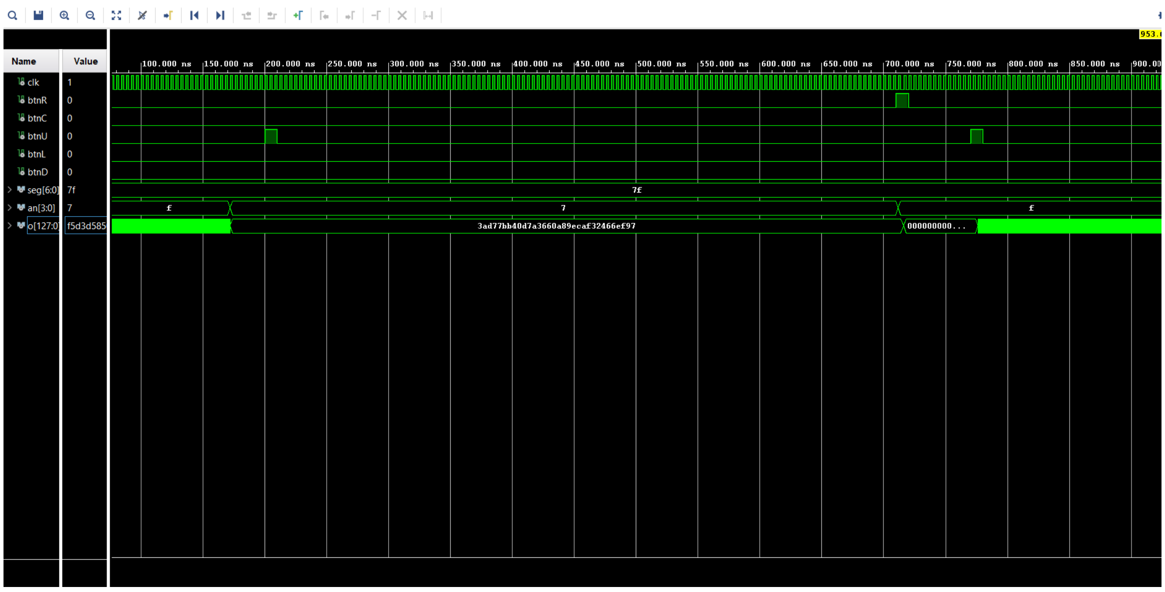


Figure 3.3: Waveform of the final module implemented part 2

can be verified accordingly.

Figure 3.4 illustrates that when the ’btnR’ is not pressed (intended for encryption restart), pressing other buttons, including ’btnC’, does not impact the system. However, upon pressing ’btnR’, the 7-segment display clears as expected. Subsequently, pressing a button to initiate encryption effectively starts the encryption process, validating the functionalities



Name	Value
clk	1
btnR	0
btnC	0
btnU	0
btnL	0
btnD	0
seg[6:0]	7f
an[3:0]	7
o[127:0	f5d3d585

Figure 3.4: Waveform of the final module implemented part 3

4

Chapter 4

4.1 Extra functionalities

4.1.1 Description

We decided add in the use of the other button. The btnU, btnL and btnD would be used to start/re-start the encryption protocol but with differnt word. The btnU, btnL and btnD would implement one of the word of the list provided. In addition to that, we added the fact that if two buttons are pressed at the same time nothing would happen and nothing would be enclanched in the module.

4.1.2 Implementation

The AES module comprises button inputs ('btnC', 'btnU', 'btnL', 'btnD', and 'btnR'), a clock input ('clk'), and output signals ('seg', 'an', 'o') representing the seven-segment display, its control signals, and the encryption output, respectively. Each button press triggers a specific behavior.

The 'process_mux' process detects button presses and generates a 'pressed' signal based on the button combinations. For instance, when 'btnC' is pressed, 'mux' gets loaded with a predefined value, setting 'pressed' to "01". Other button combinations result in different values for 'mux' and subsequently update the 'pressed' signal accordingly. The 'pressed' signal acts as an input to the FSM, influencing the state transitions.

The FSM, controlled by the 'currentState' and 'nextState' signals, orchestrates the AES encryption steps ('SB', 'SR', 'MC', 'ARK', 'RSTT', 'BTNRR'). It processes the 'pressed' signal along with its current state to determine the next state and accordingly drives the simulation. For example, in the 'RSTT' state, a 'pressed' signal of "01" initiates loading 'i4' with the value of 'mux', while other states and conditions handle subsequent AES encryption steps. Additionally, the 'BTNRR' state responds to 'pressed' being "10" (only btnR), leading to a reset of the encryption process ('nextState <= RSTT').

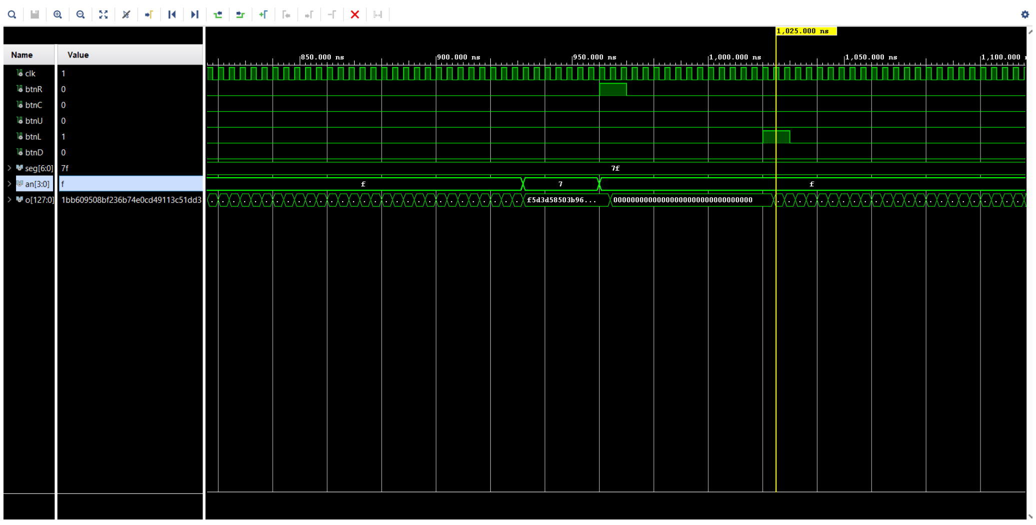


Figure 4.1: Waveform of the extra functionalities 1

4.1.3 Verification

4.1 indicates that when initiating encryption using a button other than btnC (e.g., btnL in this instance), the resulting encrypted word differs from the word encrypted by pressing btnC. Thus, for each button except btnR, there exists a unique word to be encrypted.

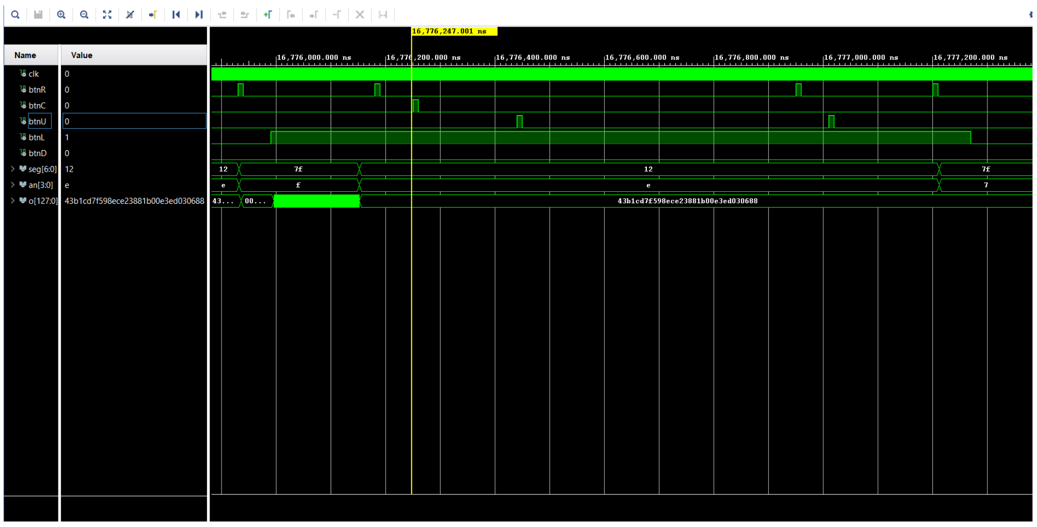


Figure 4.2: Waveform of the extra functionalities 2

4.2 demonstrates that if two buttons are pressed simultaneously, the module remains unresponsive; there will be no reaction observed from either the encryption protocol or the segment display. Here in the waveform we see that btnL is pressed for a long time and if we press btnU, btnC or btnR too, nothing will happen.