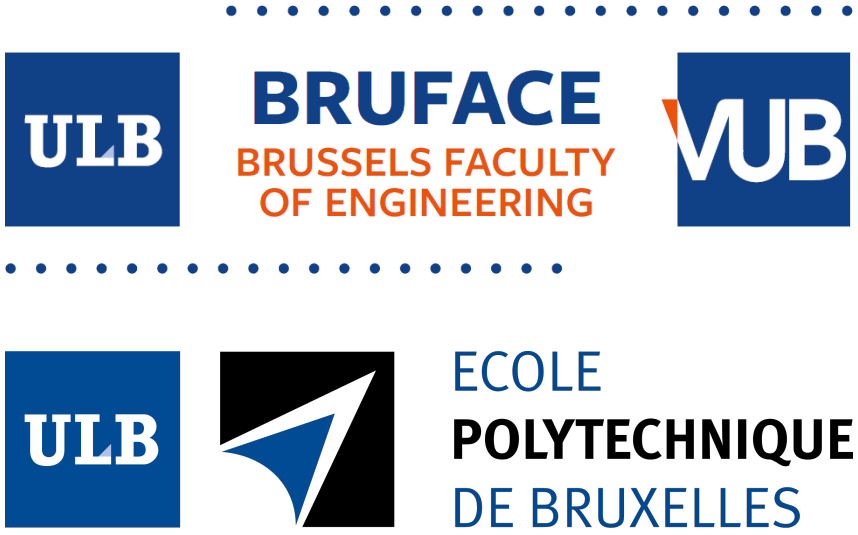

ELEC-H409

VHDL Final Project

AES encryption

Authors:

HAN Bao
JIALUN Kou

Professor:

MILOJEVIC Dragomir

Friday 23th December 2022

Contents

1 Introduction

Thanks to the efficiency of the FPGA-based implementations, FPGAs have already widely used in cryptographic application. Therefore, we will try to implement an Advanced Encryption Standard (AES) on the Basys 3 FPGA Board during this project.

AES algorithm, one of the most popular encryption algorithm of electronic data, was invented by Joan Daemen and Vincent Rijmen in 2000. As shown in figure 1.1.

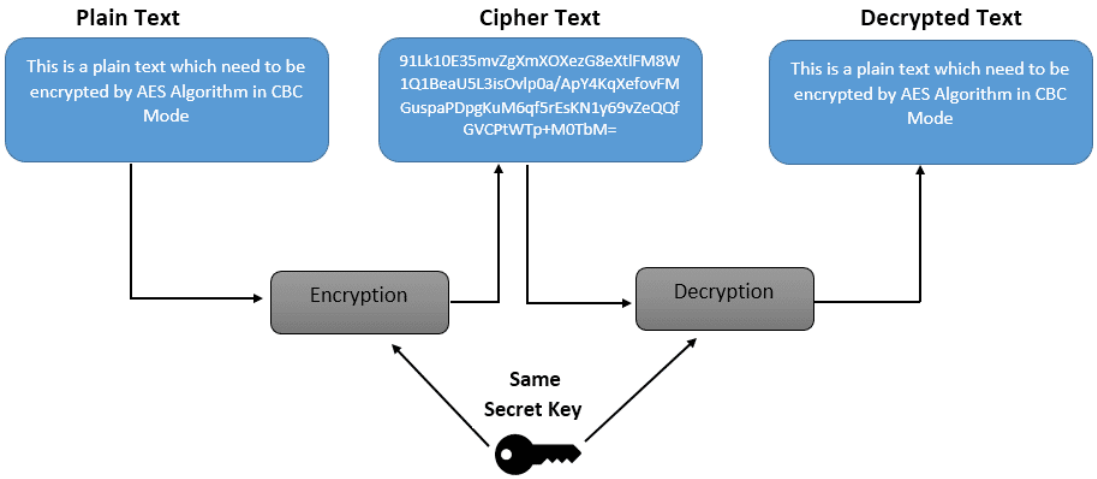

Figure 1.1: AES ENCRYPTION

In this project, the 128-bit plain-text will be converts into an 128-bit unintelligible form called cipher-text by using an 128-bit secret key. After finishing the encryption, we can still decrypt the cipher-text into a intelligible form by using the same 128-bit secret key. However, in this project, we will only focus on the encryption part of this algorithm.

The whole algorithm can be divided into four sub-functions, which are known as **Sub Bytes**, **Shift Rows**, **Mix Columns**, **Add Round Key**. These four functions will be arranged in an order.

2 Four Basic AES Encryption Steps

In this section, we will begin to implement the AES encryption algorithm. The four functions mentioned in chapter 1 will be arranged in the order shown as follow.

2.1 Add Round Key

This function can be realized by using the scheme below:

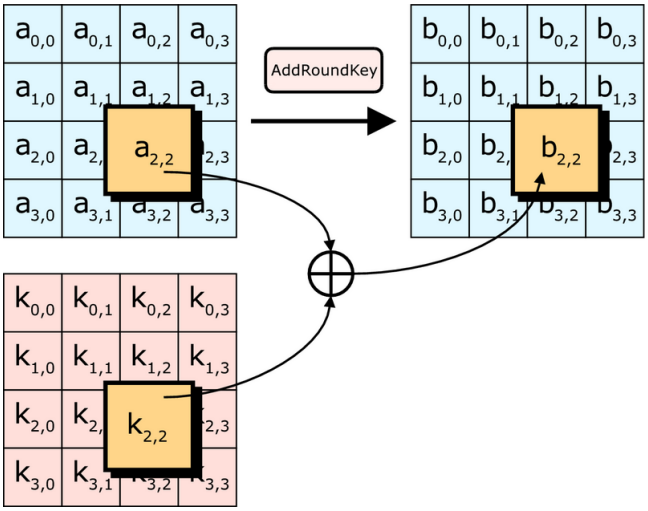

Figure 2.1: Add Round Key

The input of this module are 128 bits so we have to divided them into 16 groups and each of these group has 8 elements. The we perform XOR operation between key and the input. Then we write a textbench to verify it.

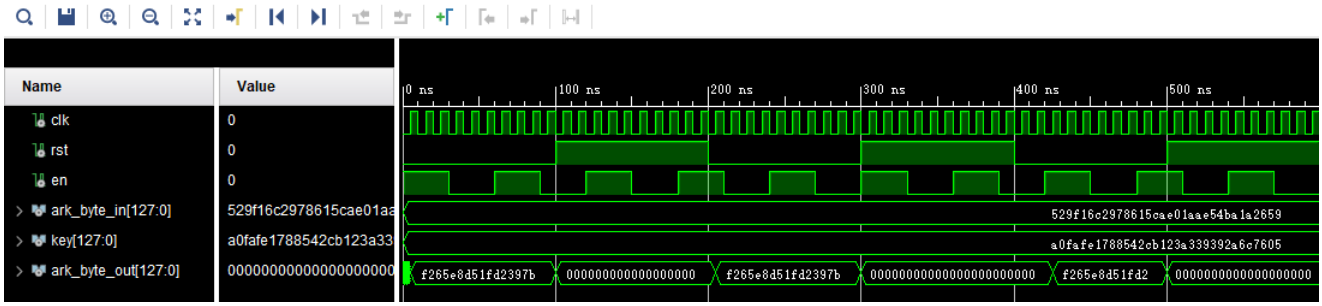

Figure 2.2: Testbench Add Round Key

Where,
ark_byte_in stands for **the plain text that you input**
ark_byte_out stands for **the output of this module**
key gets from **the key provided in the file *RoundKeys.txt***

After comparing the result we get from the testbench with the result provided by **National Institute of Standards and Technology**, we can verify that our answer totally correct.

```
MixColumn     529F16C2 978615CA E01AAE54 BA1A2659
KeyAddition   F265E8D5 1FD2397B C3B9976D 9076505C
```

Figure 2.3: Result Add Round Key

Now, the Add Round Key module is completed.

2.2 Sub Bytes

In this section, we will simply substitute each byte using a substitution table, which can be found in *S_box.vhd*. The schematic is shown below:

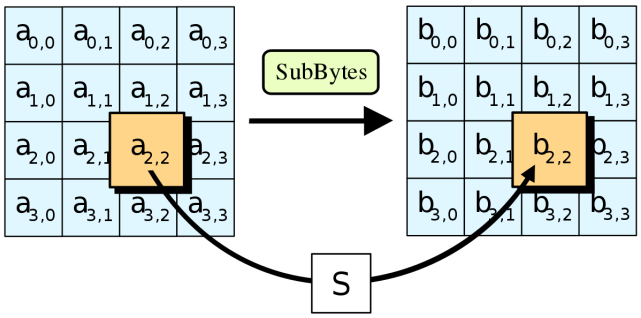

Figure 2.4: Sub Bytes

We can verify the result by writing a **testbench** and the result of the testbench is shown below:

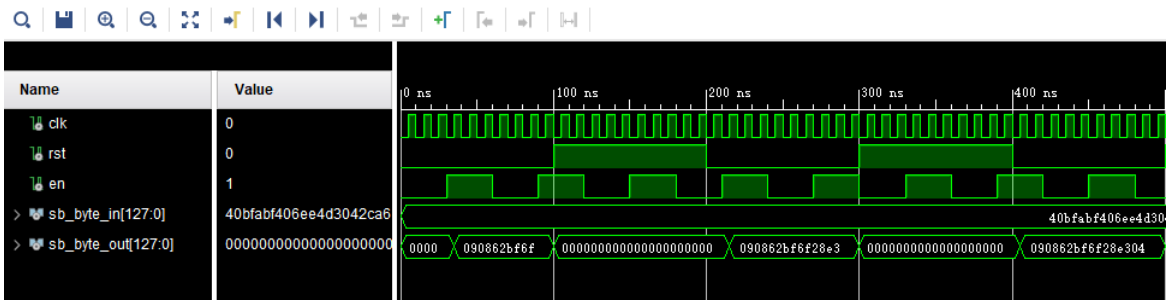

Figure 2.5: Testbench Sub Bytes

Where,
sb_byte_in stands for **the text that you input**
sb_byte_out stands for **the output of this module**
Then we compare our result with the data in *AES_Core128* shown as follow:

```
KeyAddition   40BFABF4 06EE4D30 42CA6B99 7A5C5816

Round = 1
  Substitution  090862BF 6F28E304 2C747FEE DA4A6A47
```

Figure 2.6: Result Sub Bytes

After comparing we can verify that our answer is correct therefore the Sub Bytes module is completed.

2.3 Shift Rows

In this section, we will implement a way to make the bytes in the last three rows cyclically shift to the left over a number of bytes equal to the row number as shown below.

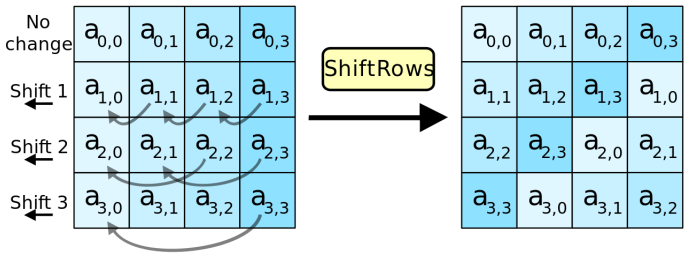

Figure 2.7: Shift Rows

The difficulty in this section is that we arranged our bytes along the column direction while the direction of the shift operation is along the row direction. And since there are only 16 blocks of bytes we decide to shift the blocks one by one.

After doing so, we can verify our thoughts by writing a testbench and the waveform of the module is seen below.

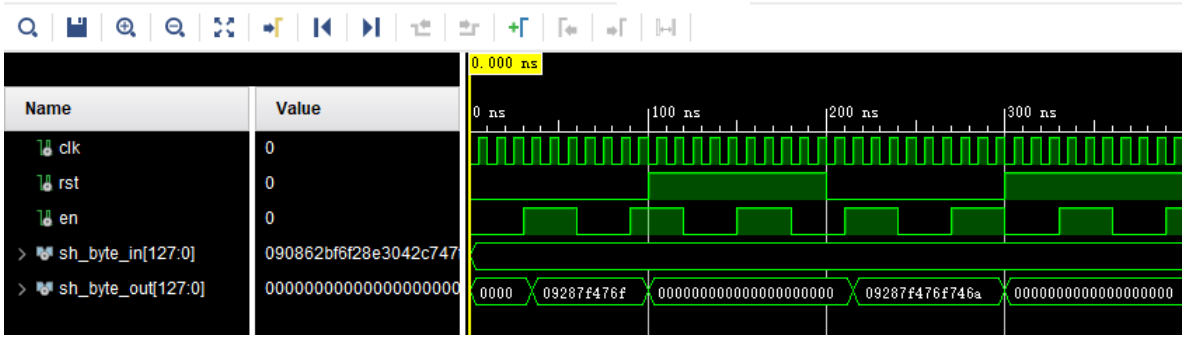

Figure 2.8: Testbench Shift Rows

Where,

sh_byte_in stands for **the text that you input**

sh_byte_out stands for **the output of this module**

Then we compare our result with the data in *AES_Core128* shown as follow:

```
Substitution  090862BF 6F28E304 2C747FEE DA4A6A47
ShiftRow      09287F47 6F746ABF 2C4A6204 DA08E3EE
```

Figure 2.9: Result Shift Rows

After comparing we can verify that our answer is correct therefore the Shift Rows module is completed.

2.4 Mix Columns

In this section, we will implement a transform to the original data by using a matrix. The schematic can be seen below

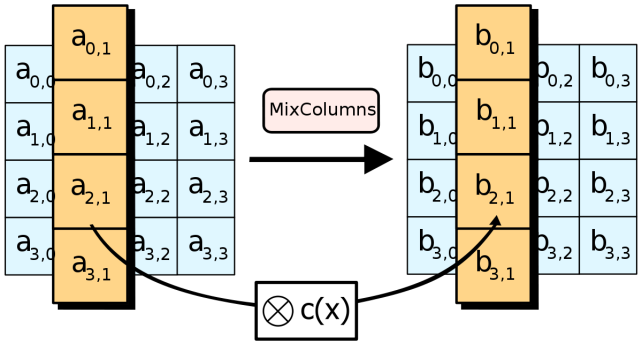

Figure 2.10: Mix Columns

With the Matrix $C(x) = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$

To realize this function, we will divide it into three sub-part shown as below.

- The first one is **XOR calculation**:

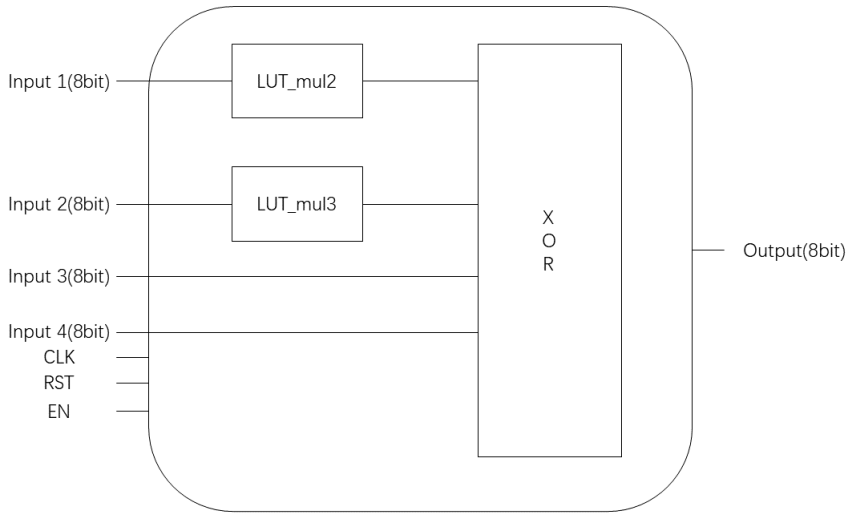

Figure 2.11: XOR calculation module

- The second one is **One Column**:

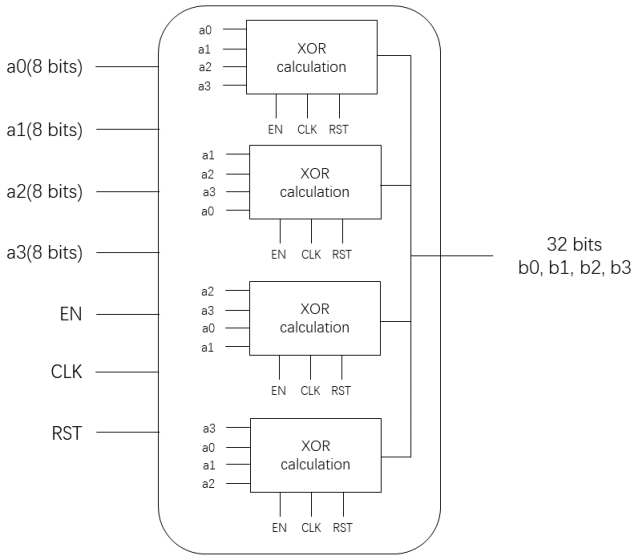

Figure 2.12: One Column module

- The third one is also known as the top level here we name it as **Mix column**:

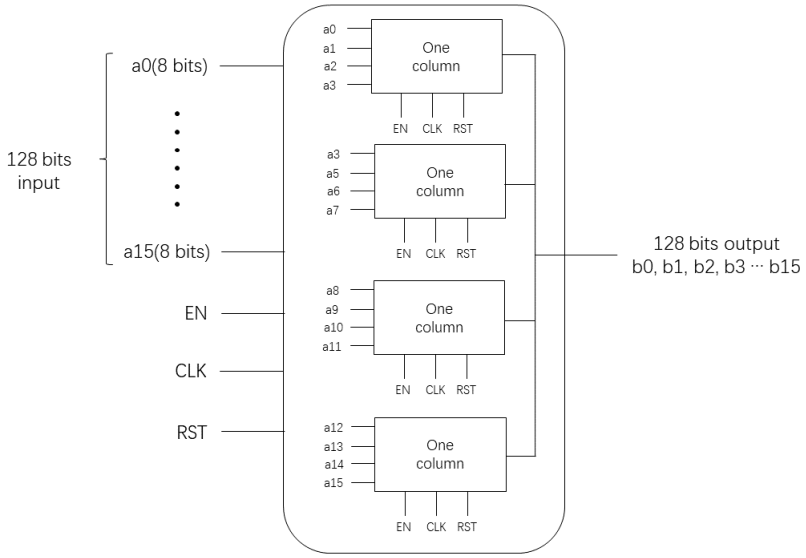

Figure 2.13: Mix column module

The hierarchy relationship of this three part can be clearly shown as below

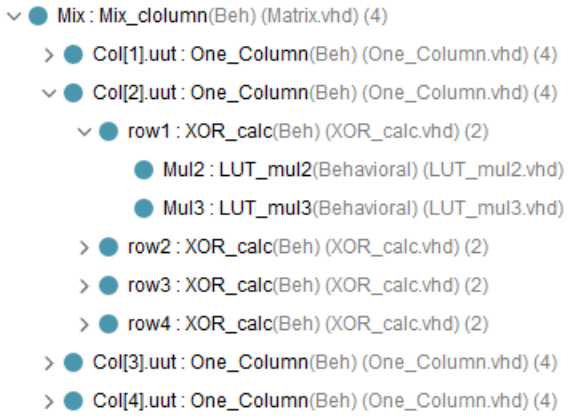

Figure 2.14: hierarchy relationship

Now, to verify our design, we could write a testbench and compare the result we derive with the data provided in file *AES_ Core128*

Below is the result we get from the testbench:

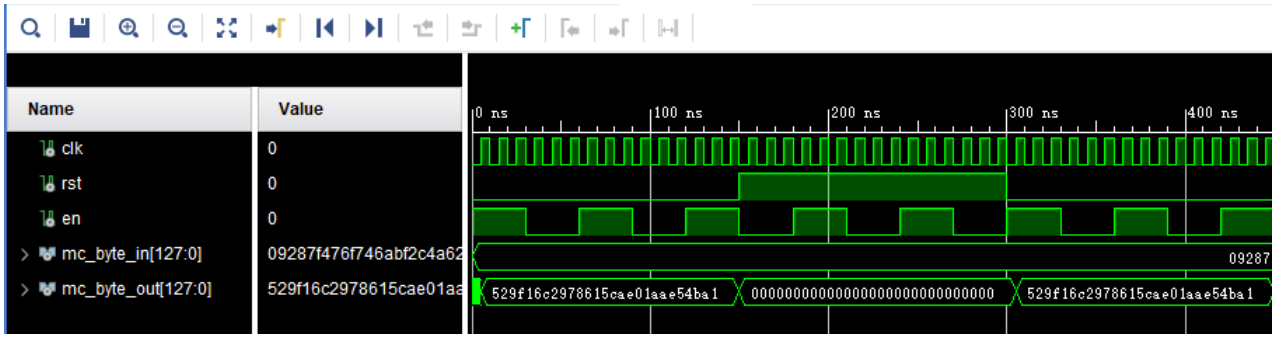

Figure 2.15: Testbench Mix Column

Then we compare the result above with the data in the file, as seen below.

```
ShiftRow      09287F47 6F746ABF 2C4A6204 DA08E3EE
MixColumn     529F16C2 978615CA E01AAE54 BA1A2659
```

Figure 2.16: Result Mix Column

After comparing we can verify that our answer is correct therefore the Mix Column module is completed.

3 New Model Based on Four Basic Steps

In this section, we will implement some higher level functions based on the four functions we realized in the chapter 2.

Also known as

- Add Round Key
- Sub Bytes
- Shift Rows
- Mix Columns

3.1 Three Steps Module

This model consists of *Sub Bytes*, *Shift Rows* and *Add Round Key*. The overall structure is shown as below:

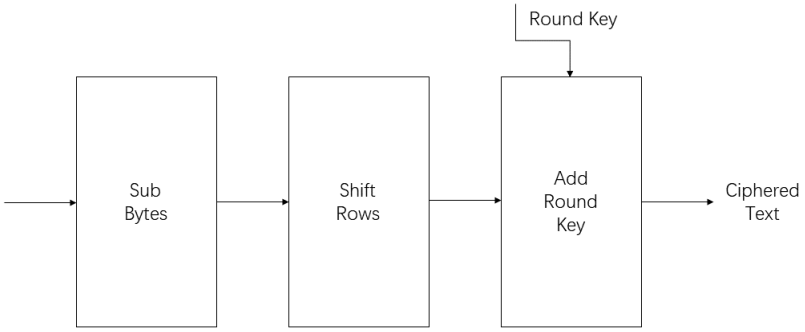

Figure 3.1: Three Steps

The module has five input with only one output, we define the port of this entity as follow:

```
entity Three_step is
Port (
        clk, rst,en          : in STD_LOGIC;                --clock, reset and enable
        byte_in              : in STD_LOGIC_VECTOR(127 downto 0);   --input 128 bits
        key                  : in STD_LOGIC_VECTOR(127 downto 0);   --key 128 bits
        byte_out             : out STD_LOGIC_VECTOR(127 downto 0)   --output 128 bits
 );
end Three_step;
```

To realize each AES step executed in one clock cycle, we design a state machine. As shown in diagram, this model contains 3 states (Sub Byte, Shift Row, Add Key), and the next state was triggered by the rising edge of clock.

Then to verify our design, we write a testbench to test our design. And the stimulation we give to the entity can be seen as follow:

```
    clk_proc : process  --to control the clock signal
        begin
            clk <= NOT clk;
            wait for 5 ns;
        end process;

    RST_proc: process   --to control the reset signal
        begin
            rst <= NOT rst;
            wait for 100 ns;
        end process;
    en_proc : process   --to control the enable signal
        begin
            en <= NOT en;
            wait for 50 ns;
        end process;

    SB_porc: process    --to control the input signal
        begin
            test_byte_in <= x"BB36C7EB88334D49A4E7112E74F182C4";
            key          <= x"d014f9a8c9ee2589e13f0cc8b6630ca6";--"00101011";
            wait for 100 ns;
        end process;
```

After test, we get the waveform of the Three Step module.

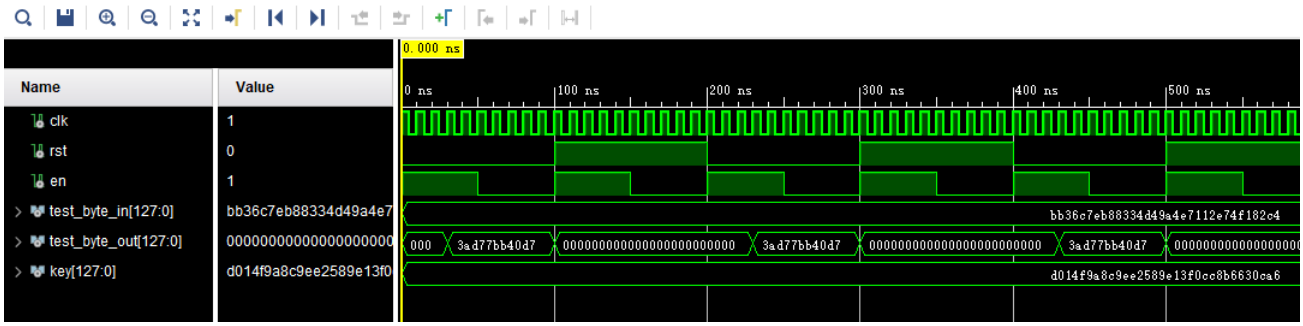

Figure 3.2: Testbench Three Steps

Then we compare the result above with the data in the file, as seen below.

```
  KeyAddition   BB36C7EB 88334D49 A4E7112E 74F182C4

Substitution  EA05C6E9 C4C3E33B 49948231 92A1131C
ShiftRow      EAC3821C C49413E9 49A1C63B 9205E331
KeyAddition   3AD77BB4 0D7A3660 A89ECAF3 2466EF97
```

Figure 3.3: Result Three Steps

As shown in test bench waveform, the result of each component matches the value of test vector, and each AES step perfectly executed in one clock. this model used Three clock cycles in total. Therefore, the Three Steps module is completed.

3.2 Four Steps Module

This model consists of *Sub Bytes*, *Shift Rows*, *Mix Columns* and *Add Round Key*. The overall structure is shown as below:

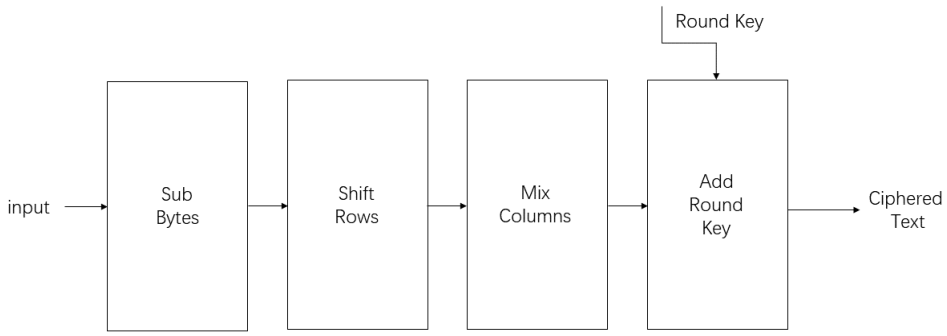

Figure 3.4: Four Steps

The module has five input with only one output, we define the port of this entity as follow:

```
entity Four_step is
Port (
        clk, rst,en          : in STD_LOGIC;                 --clock, reset and enable
        byte_in              : in STD_LOGIC_VECTOR(127 downto 0);   --input 128 bits
        key                  : in STD_LOGIC_VECTOR(127 downto 0);   --key 128 bits
        byte_out             : out STD_LOGIC_VECTOR(127 downto 0)   --output 128 bits

 );
end Four_step;
```

This model contains 4 states, (sub byte, shift row, mix column, add key) the next state was triggered by the rising edge of clock., this model used four clock cycles in total.

Then we write a testbench to verify our design. The waveform we obtained from the testbench is seen as follow:

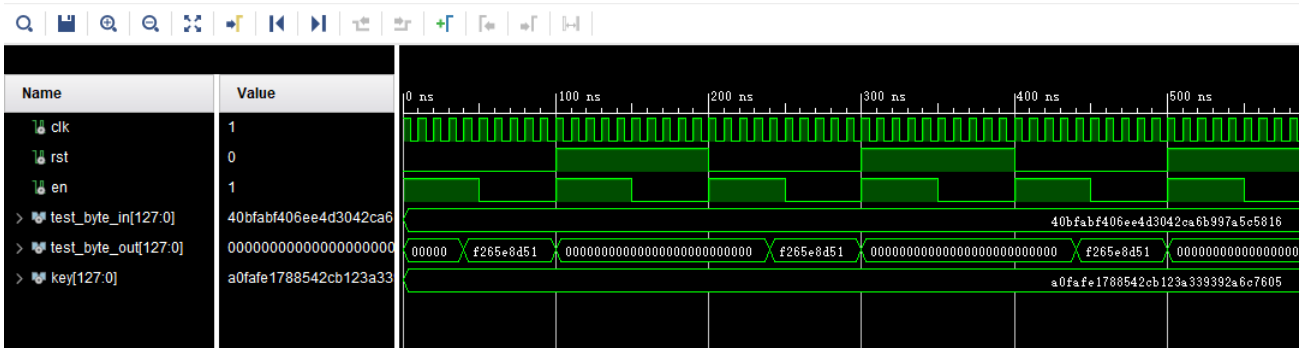

Figure 3.5: Testbench Four Steps

Then we compare the result above with the data in the file, as seen below.

```
KeyAddition   40BFABF4 06EE4D30 42CA6B99 7A5C5816

Round = 1
  Substitution  090862BF 6F28E304 2C747FEE DA4A6A47
  ShiftRow      09287F47 6F746ABF 2C4A6204 DA08E3EE
  MixColumn     529F16C2 978615CA E01AAE54 BA1A2659
  KeyAddition   F265E8D5 1FD2397B C3B9976D 9076505C
```

Figure 3.6: Result Four Steps

After comparing we can verify that our answer is correct therefore the Four Steps module is completed.

3.3 Nine Loops Module

This model consists of nine *Four Steps Model*. The overall structure is shown as below:

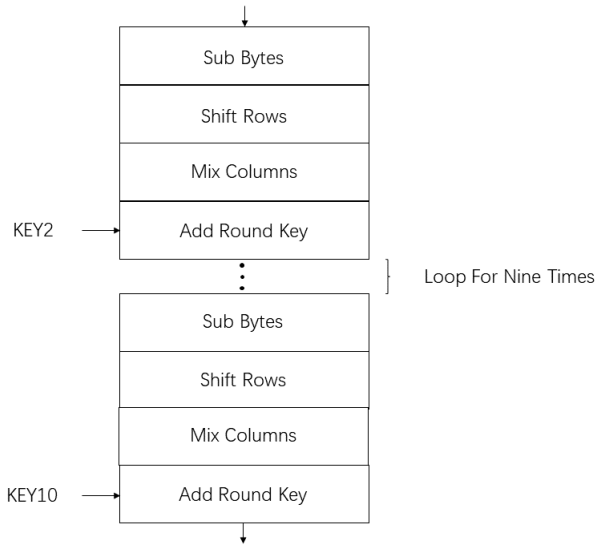

Figure 3.7: Nine Loops

The module has four input with only one output. And this model contains 9 states (loop1 ...loop9), each state changes every four clock cycles, therefore this model used 36 clock cycles in total.

To verify our thoughts, we write a testbench to test our entity. The result of the testbench is shown as below:

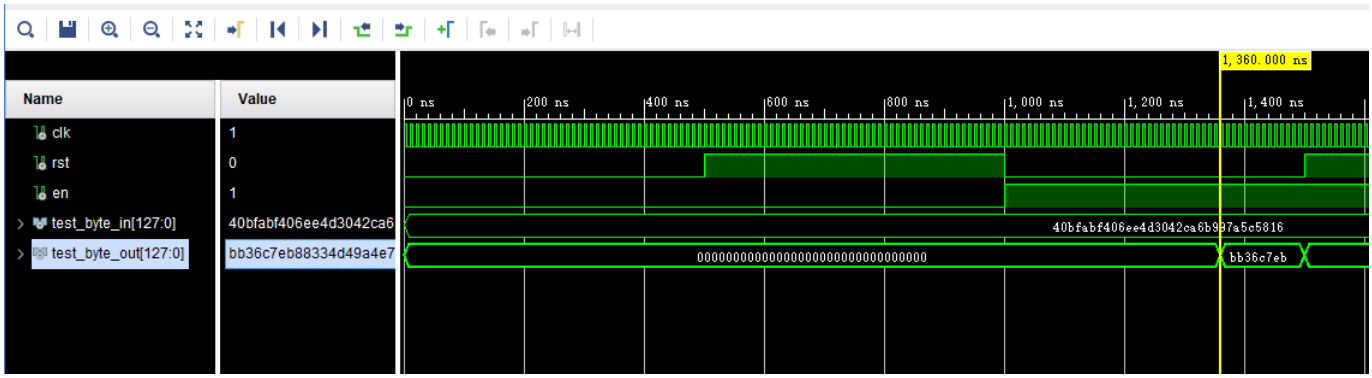

Figure 3.8: Testbench Nine Loops

Here, according to the mark we have made in the waveform. The *en* signal begins at 1000 ns and after nine times of the loop, the correct output should begin at 36 clock cycles after the *en* signal becomes *1*. In other words, the correct output should begin at 1360 ns (since for one clock cycle, the period is 10 ns).

Having checked the clock, we compare the result derived from the testbench with the data in the file.

```
Round = 9
  Substitution  830EB4ED FF338109 C115F05E 7791A6AF
  ShiftRow      8333F0AF FF15A6ED C191B409 770E815E
  MixColumn     1741A118 91C99168 8C36386F 23AD82AA
  KeyAddition   BB36C7EB 88334D49 A4E7112E 74F182C4
```

Figure 3.9: Result Nine Loops

After comparing we can verify that our answer is correct therefore the Nine Loops module is completed.

4 AES Algorithm and Led Control

In this section, we will implement AES algorithm by using the model we created in the previous chapter. Moreover, LED display unit and an additional function realization model will be also created.

4.1 AES Realization

This model consists of three different models created in the previous chapters, they are**Add Round Key**, **Nine Loops** and **Three Steps** respectively. And its structure is shown below:

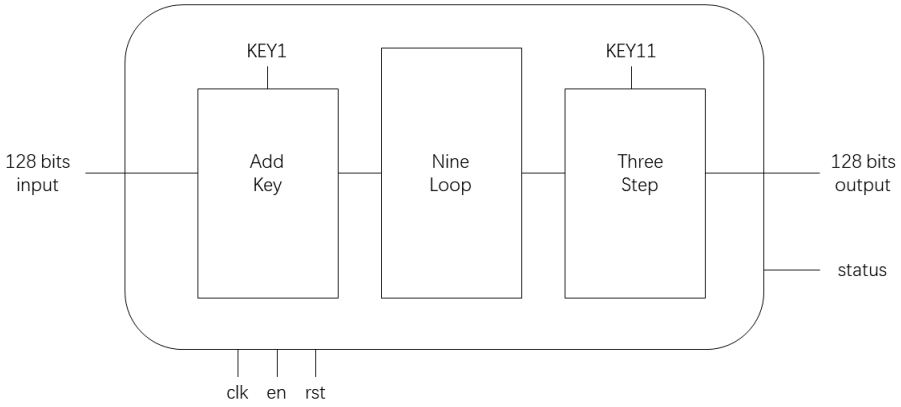

Figure 4.1: AES module

As we can see from the structure above, this module has 4 input and two output. And the *status* output will be connected to the module that realize a new function. This model contains three states (Add round key, Nine loop and three step), using 40 clock cycles in total.

Then we write a testbench to verify our design. The result of the testbench is shown below:

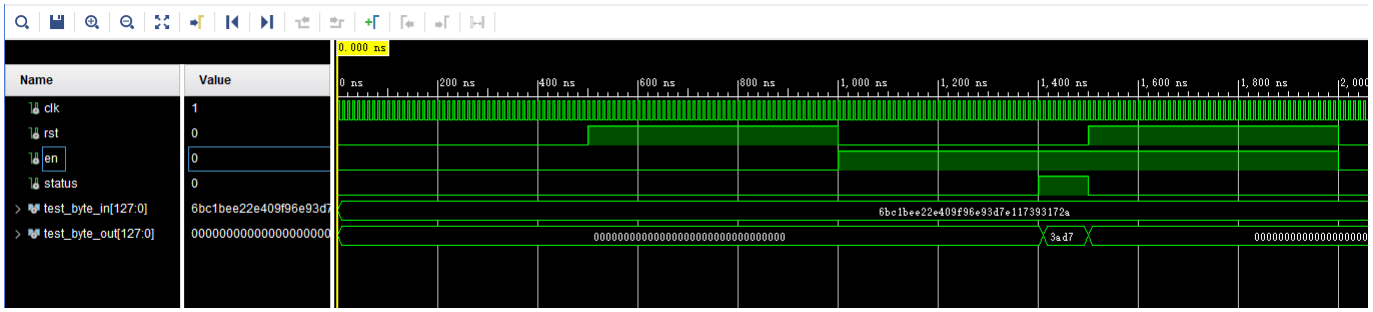

Figure 4.2: Testbench AES

Here we can see that

- When *en* is 0, then AES module does not work so the *byte_ out* and *status* are 0;
- When *en* is 1 and *rst* is 0, then AES finishes calculation after 400 ns (40 clock cycles with each clock cycle lasts for 10 ns) . In this situation, *status* becomes 1 and *byte_ out* shows the correct result.
- When *en* is 1 and *rst* is 1, then AES restart. In the waveform we can see that after the *rst* becomes 1 then both *byte_ out* and *status* are set to 0.

Therefore we can say that the logic of the circuit we design is correct. Then we use the data in the given file to check if the output of this module is correct, as seen below:

```
Ciphertext is
    3AD77BB4 0D7A3660 A89ECAF3 2466EF97
    F5D3D585 03B9699D E785895A 96FDBAAF
    43B1CD7F 598ECE23 881B00E3 ED030688
    7B0C785E 27E8AD3F 82232071 04725DD4
```

Figure 4.3: Result AES

After comparing we can verify that our answer is correct therefore the AES module is completed.

4.2 LED Display

In this section, we will design a module to correctly control the Basys-3 Boards. Fisrt we add the constraints file and un-comment the I/O we want to use. Then, in order to display the four 7-segment "at the same time", we will let each digit illuminated for just one-fourth of the time, and let it update faster than the human eye can detect. So we write a FSM with a new clock whose cycle is 4 ms.

Then we write a testbench and the result shows as below:

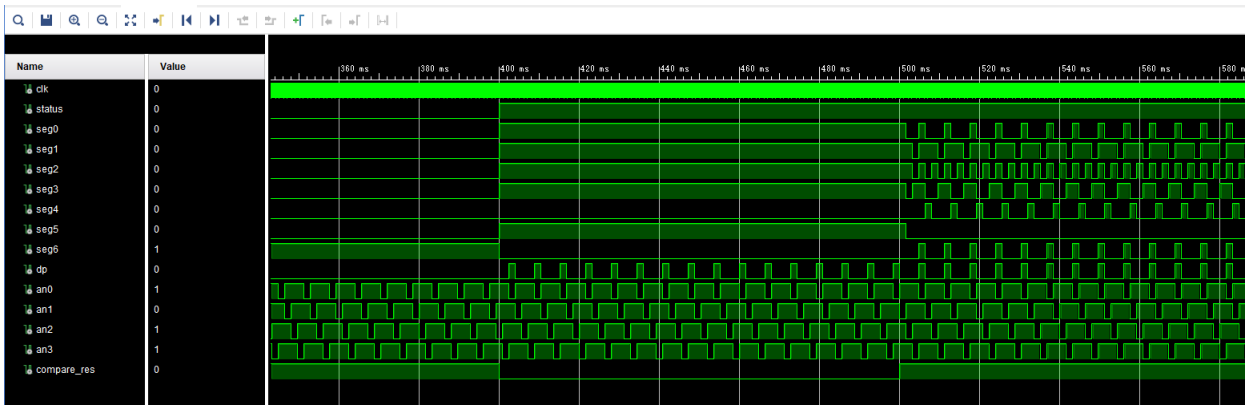

Figure 4.4: Testbench LED

We can see the each *an* only activates for one-fourth of the new clock cycles we create, which means that the update rate can be faster than the human eye can detect.

The function of the module can be divided into three parts shown below:

- When both *status* and *compare_res* is 1. It means that the encryption has finished and the result we get is correct. And FPGA Board will show "A.E.S.1." on its LED digit.
- When *status* is 1 while *compare_res* is 0. It means that the encryption has finished but the result we get is wrong. And FPGA Board will show "mirrored 7" on its LED digit.
- When *status* is 0 then no matter what *compare_res* is. It means that the encryption is unfinished. And FPGA Board will show "0000" on its LED digit.

To further verify our thoughts, we can run our module on the FPGA Boards, as shown below.

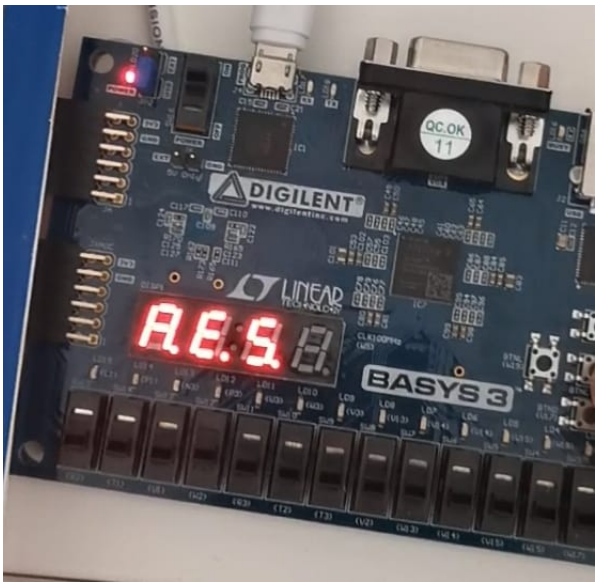

Figure 4.5: LED on Board

Here, we can see that four LED digit illuminate "at the same time". Therefore, the LED Display module is completed.

4.3 Additional Function: Compare Module

Here we create a module to realize additional function. The new function is as below:

- This is a module with 2 input(AES ouput and clk) and 1 output(compare_res).
- We set the correct answer in our module in advance. After we run all the module, it will automatically compare the result we calculated with the correct answer we pre-set. If the result we get by our algorithm is false, than the output of Compare module will become 0, otherwise it will become 1.

Since this module is relatively easy to realize, we are not going to show the waveform here.(We have tested it and the result show this module is properly working).

So far, we have completed all the sub modules and in the next chapter, we will build the top level of this project.

5 Top Level and Verify

In this chapter, we will build the top level of this AES project.

5.1 Structure

Before implement the top level, it is highly important to come up with a proper block diagram. And the structure we build for this top level can be seen below:

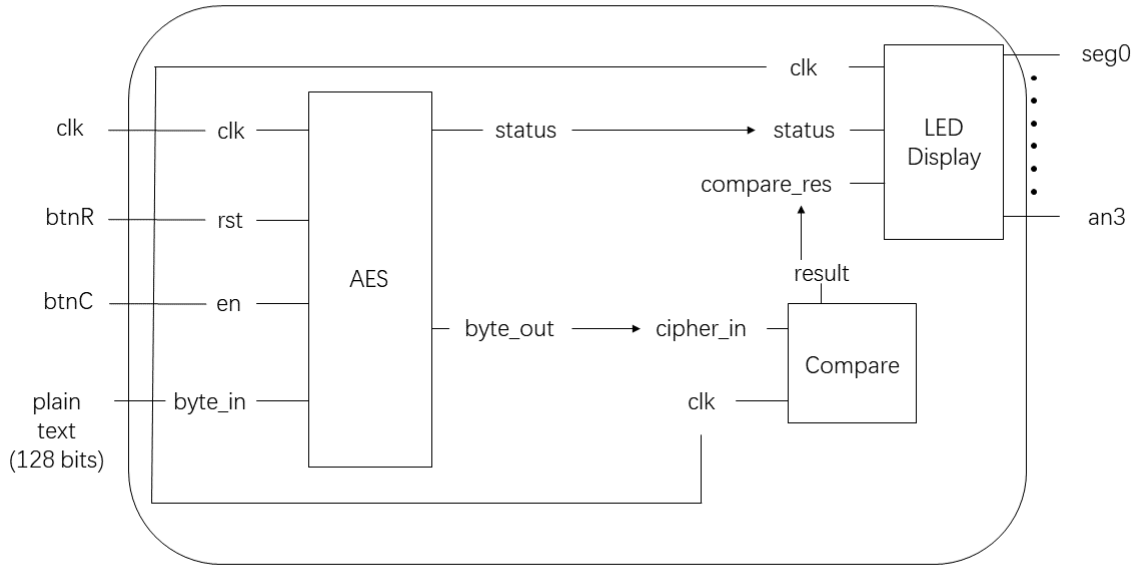

Figure 5.1: Top

And the hierarchy structure is shown below:

Figure 5.2: Hierarchy of Top Level

After connecting each port of each module, we can first write a textbench to check if the board can run properly. The interconnection of each module is shown as below:

```
signal states : std_logic;
                            --signal between AES_state and LEDdisplay_status
signal compare: std_logic;
                            --signal between compare_result and LEDdisplay_compare_res
signal ciphered:std_logic_vector(127 downto 0):=(others => '0');
                            --signal between AES_byte_out and compare_result
signal plain_text: std_logic_vector(127 downto 0):=x"6BC1BEE22E409F96E93D7E117393172A";

begin
uut1: AES          port map(clk => clk, rst => btnR, en => btnC,
                            byte_in => plain_text,
                            byte_out => ciphered, status => states);
uut2: led_display  port map(clk => clk, status => states,
                            compare_res => compare,
                            seg0 => seg0, seg1 => seg1, seg2 => seg2, seg3 => seg3,
                            seg4 => seg4, seg5 => seg5, seg6 => seg6,
                            dp => dp,
                            an0 => an0, an1 => an1, an2 => an2, an3 => an3);
uut3: compare_text port map(clk => clk, cipher_in => ciphered, result => compare);
```

5.2 Testbench

Here we write a testbench and according to waveform we obtained, this board can run properly. The testbench of this module is shown as below:

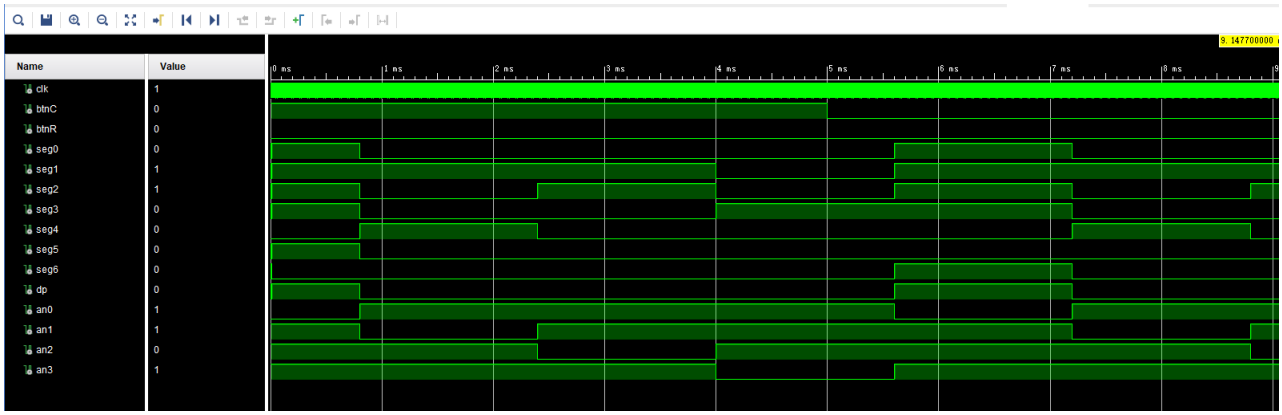

Figure 5.3: Testbench Top

5.3 Verifying on Board

Then we use our FPGA Board to berify our design.

- First we press the button in the middle to enable, then the board shows "A.E.S.1", which means that the encryption has finished and the result is correct.

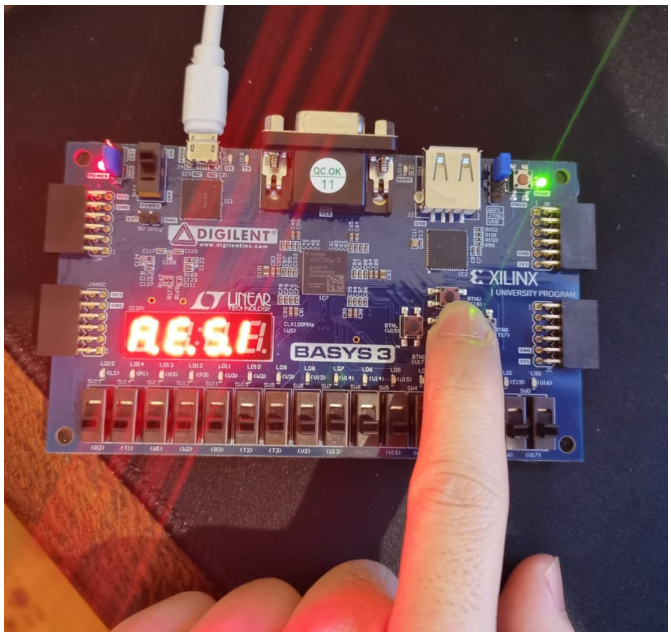

Figure 5.4: Finished and Correct Result

- Then we press the button on the right, which means that the board is in restart, therefore the encryption does not finished. Then the board shows "0.0.0.0"

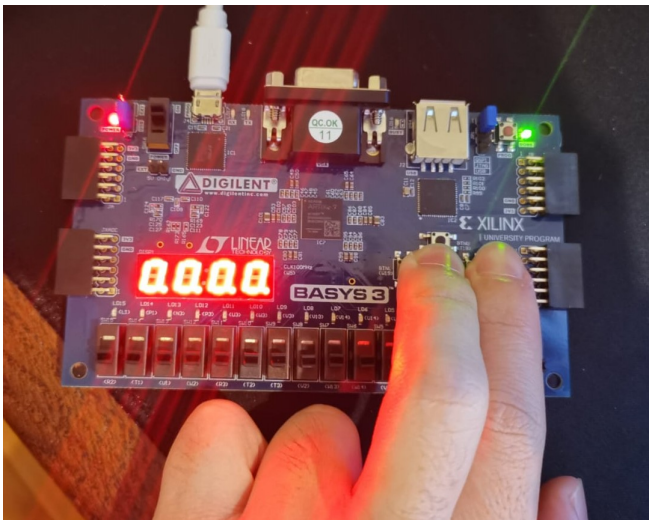

Figure 5.5: Unfinished

- After that we release the button on the right then the board start working again and get the correct result, shown as"A.E.S.1".

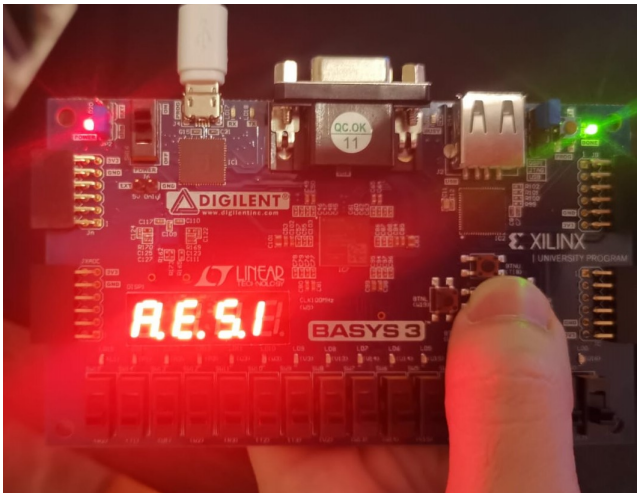

Figure 5.6: Finished and Correct Result Again

- At last we only press the right button and the result we obtain surely does not match to any of the result in file *AES_Core128*. Thus the LED digit shows so-called "mirrored-7".

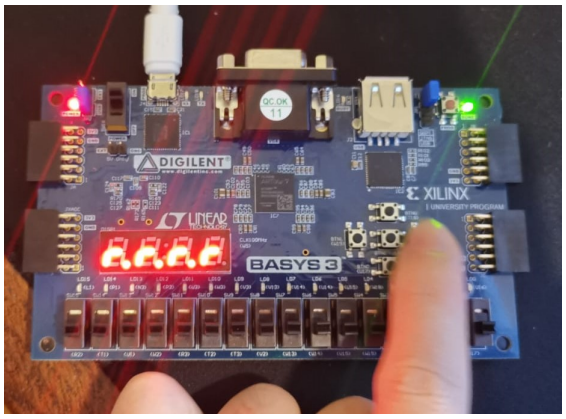

Figure 5.7: Finished and Wrong Result

So far, we have successfully implemented AES on Basys 3 FOGA Boards and verified the result.