COL215 Hardware Assignment 3 Part II:

Ayush Ahuja: 2023CS50911

Meet Patil: 2023CS50713

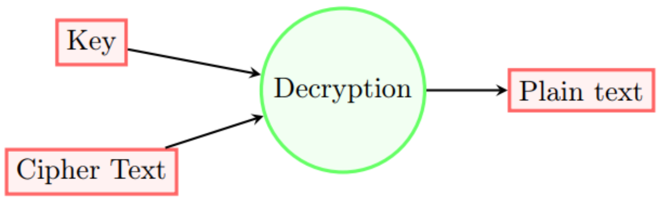
1. Objective:

The objective of the assignment is: implementation of AES decryption operation. The components involved are memory elements (RAM, ROM and registers) and logical unit.

2. Problem Description

Given a cipher text and a key, the task is to perform an AES decryption operation to display plain text on 7-seven display.

Figure 1: Overview of task: Decryption



• Input cipher text is of the size in multiples of 128 bits and will be provided in the COE file (8 bit binary unsigned). The input file should be stored in 1-D array format in the block RAM, starting from address 0 (000016) in row major format.

• Key will be provided via the COE file. All the round keys will be provided in the COE file.

3. Design Decisions:

In **part 1** of the assignment, we did the following tasks:

1. Initialized BRAM for storing the cipher text.
2. Made 4 design sources for the 4 different operations.
3. Made a displaytext design source, which displays 4 8 bit vectors at a time.

Now for **part 2** of the assignment, our design decisions were:

2 tasks were needed to be done:

1. Making the compute unit which is the FSM, and controls the read write and the 4 operations.
2. Displaying the plain text characters in a scrolling manner.

The FSM:

We defined a register *total_reg* which is a 128-bit register, which is used in the states to perform operations on a 128-bit part of the cipher text.

For the 4 states which we defined for the 4 different operations, we have defined 3 new variables each, for enable_op, done_op and counter_op.

Enable is an indicator in the fsm indicating that correct values are ready for giving input into the component performing the operation, and we can now enable the component for performing the operation with correct input values.

Done is an indicator of when the operation of the whole state is over. The components we have defined don't take 128-bit values, but the state has to perform operations on 128-bit values, so when the whole 128-bit register is processed, done indicates that transition from this state can be initiated now.

Also, within each state, the component takes some time to compute. So how much to wait is controlled by the variable counter.

There is a total of 8 states in the FSM which are as described below:

1. counting: This state serves as the default starting and fallback state. It initializes the FSM before specific operations are activated, allowing various components to complete their tasks. Once the required conditions are met, the FSM transitions to one of the operational states.

2. xor_op: This state performs an XOR operation between elements from the input data (stored in tot_reg) and round keys fetched from the ROM. The XOR is executed byte-by-byte. For each byte, the ROM address is calculated, and the byte data is read. When a full 16-byte round is completed, the state marks the XOR operation as done and resets counters to prepare for the next state.

3. shiftrow_op: This state applies the inverse ShiftRows transformation on tot_reg data. Each row in the data block is shifted by a specified offset based on its position, undoing the shifting applied during encryption. The state iterates over each row (using the row_index signal), performs the shift, and then stores the shifted row back in tot_reg.

4. mixcol_op: This state performs the inverse MixColumns operation, processing data in tot_reg in 4-byte columns. Each column undergoes matrix multiplication in GF(2^8), modifying the column values according to the inverse MixColumns transformation. After each column is processed, the results are stored in tot_reg, and the state transitions once all columns are processed.

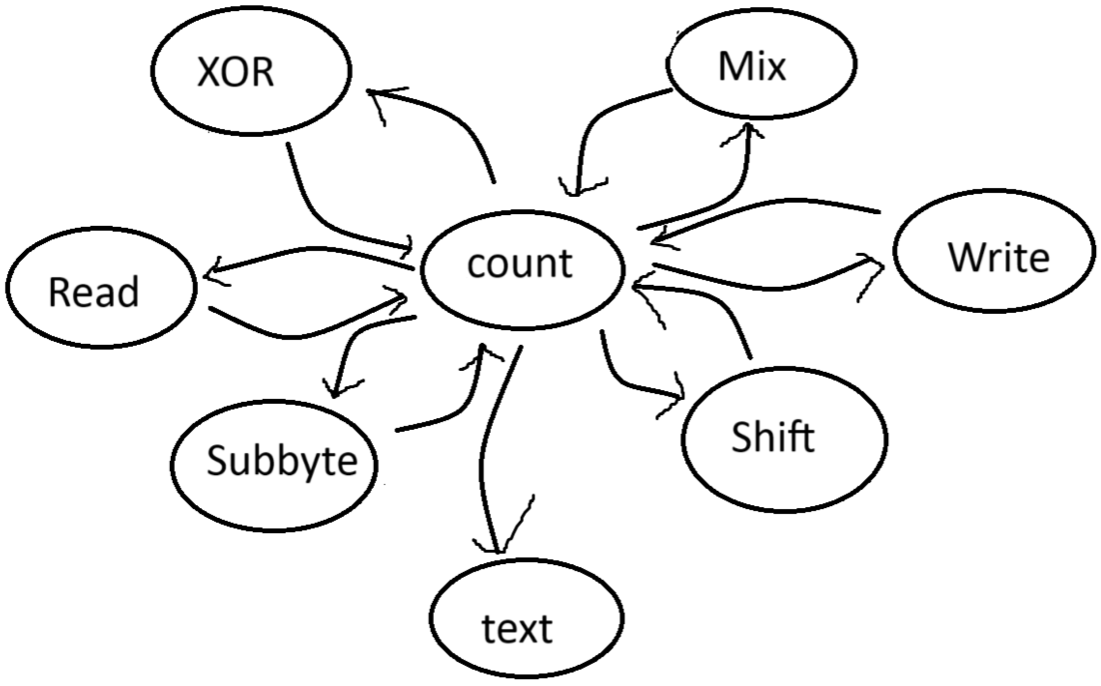
5. subbyte_op: This state performs the inverse SubBytes operation, which substitutes bytes in the tot_reg data using a lookup table (invS-box). The invSubBytes component is enabled, and the FSM iterates over each byte, applying the substitution and storing the result back in tot_reg. The state transitions once all 16 bytes have been processed.

6. plain_text: This state manages plaintext display output. It is used to display specific bytes from the processed data on an output display using the displaytext component. It cycles through each byte in the plaintext data and displays it on the hardware interface by updating the digit signals. The FSM remains in this state until the display counter finishes its cycles.

7. reading: This state reads data from the Block RAM (BRAM). In this state, bram_access components are enabled for reading specific addresses. This data is then loaded into the FSM's registers for further processing. The state cycles through addresses and reads bytes sequentially until all required data is fetched.

8. writing: This state writes processed data back to the BRAM. Similar to the reading state, but in reverse, it cycles through addresses and writes data from the FSM's registers into memory. Once all required bytes have been written, the state signals completion, transitioning back to counting or moving to another operational state.

New design file we created:
We created a ROM file which stores the round keys.



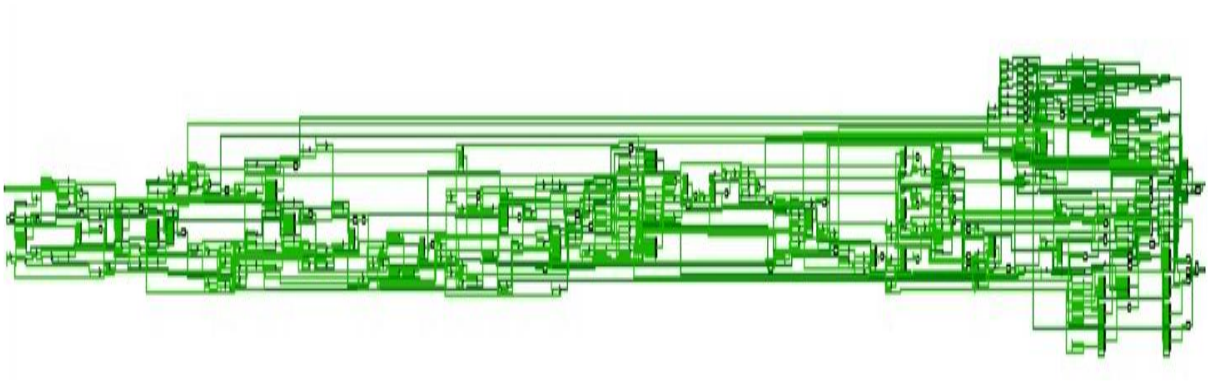
Effective state diagram for 128-bit part, with appropriate transition conditions33

Giving proper delays after operations:
As mentioned earlier, we use variables named counters, which decrease by 1 with each clock cycle. Until they get to 0, the operation is let to go on, which effectively acts like wait. Now, for this, we have to initialize counters with proper values. For choosing the values, we made observations from the simulations we did in part 1.

=> What if cipher text is not just 128, but a multiple of 128?
For handling this, we have made a variable multiple_count, in which we have to input (length of cipher text)/128. So for each 128-bit part of the cipher text, we redo all the operations above, and rewrite the output in the corresponding location in the BRAM.

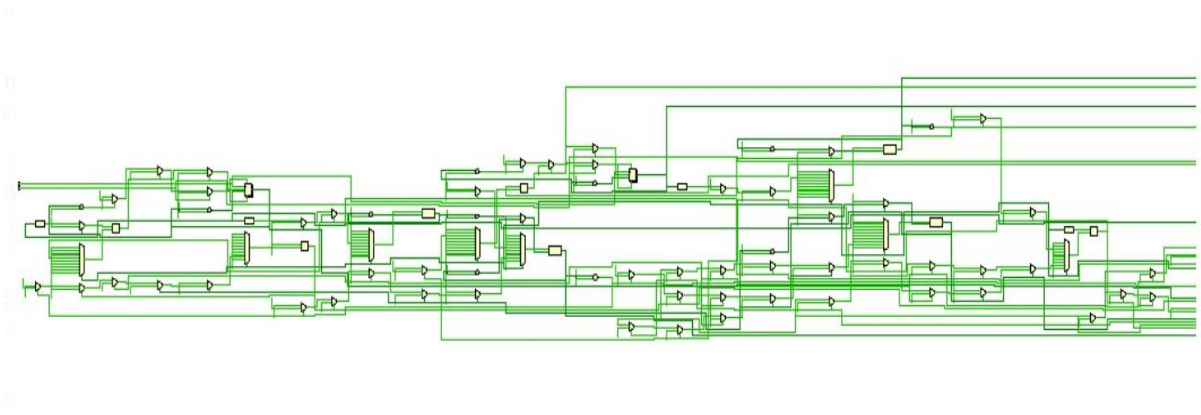
4. Block Diagram:

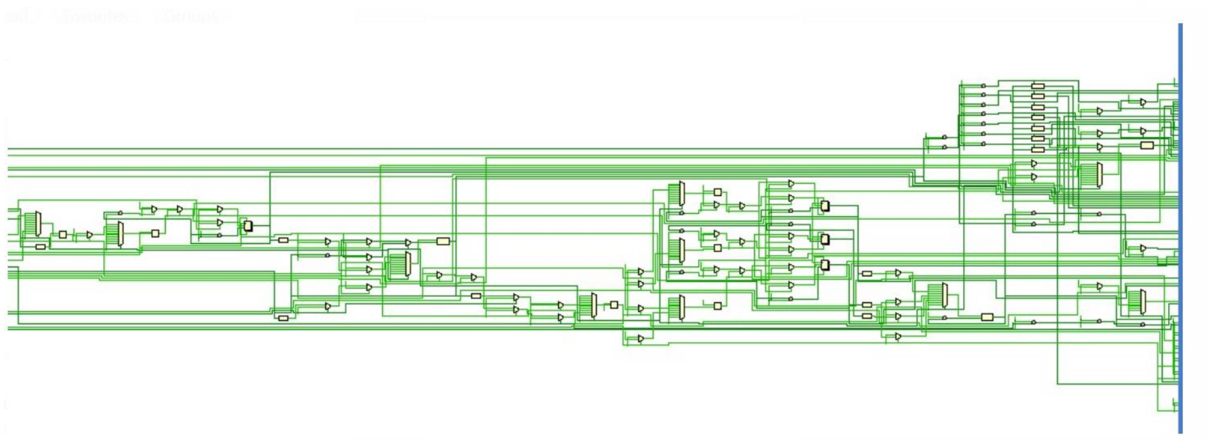
FSM:

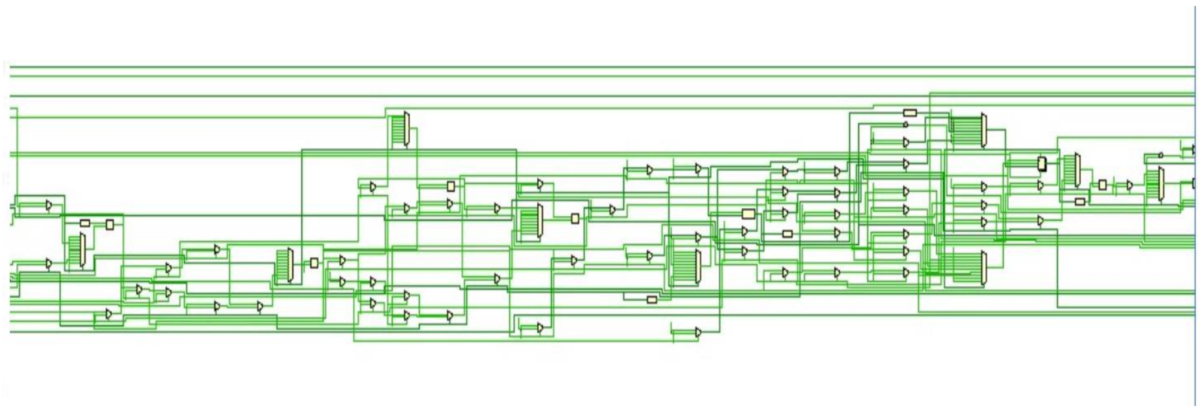


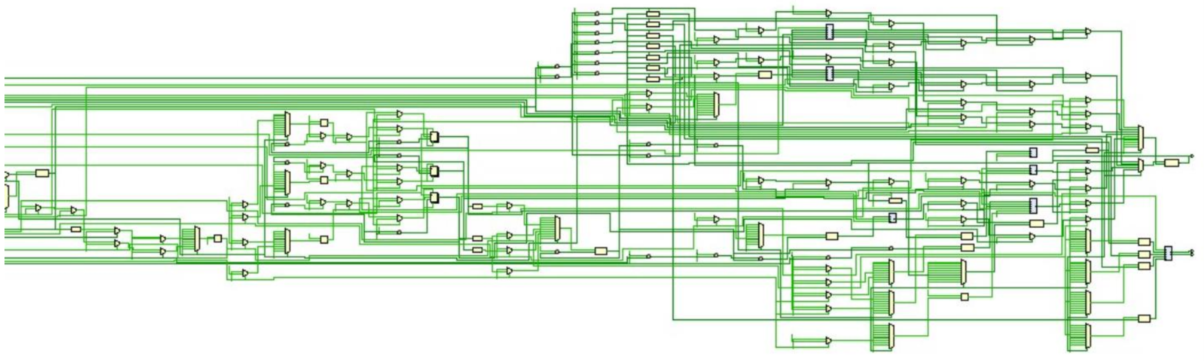
Overall diagram

Zoomed parts:



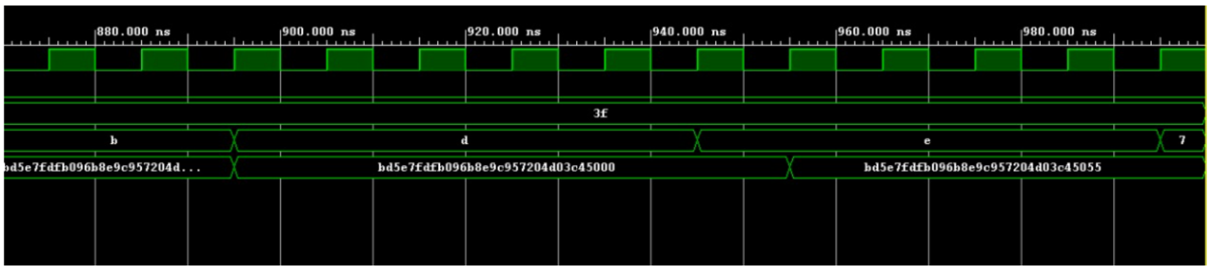






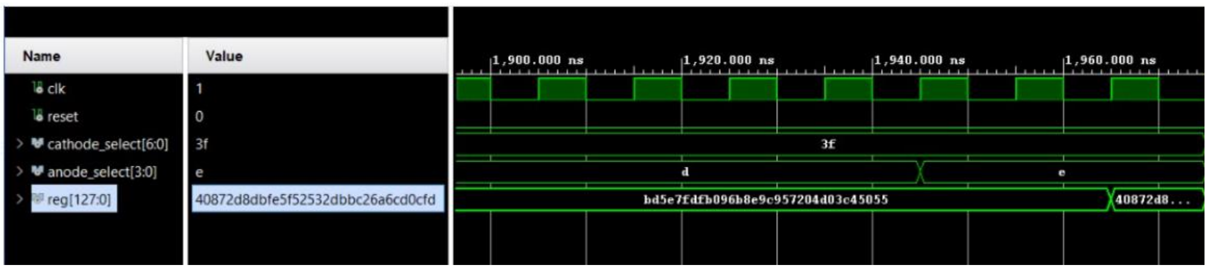
5. Simulations:

Reading:



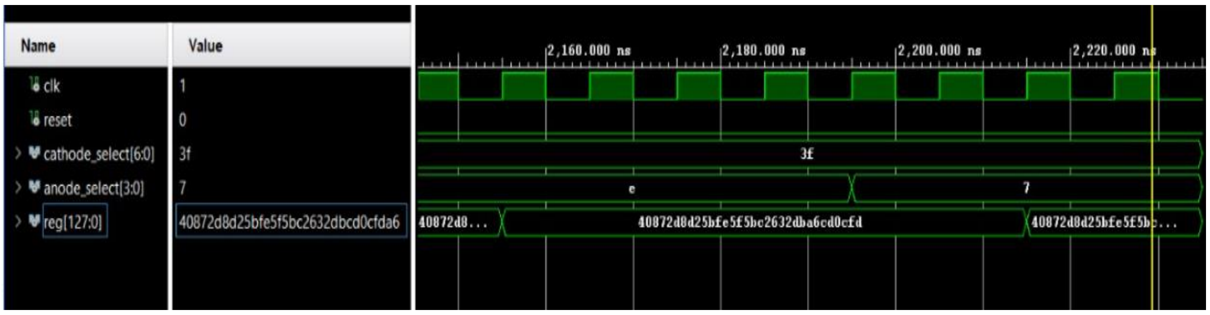
```
880.000 ns    900.000 ns    920.000 ns    940.000 ns    960.000 ns    980.000 ns
3f
b    d    e    7
bd5e7fdfb096b8e9c957204d...    bd5e7fdfb096b8e9c957204d03c45000    bd5e7fdfb096b8e9c957204d03c45055
```

XOR:



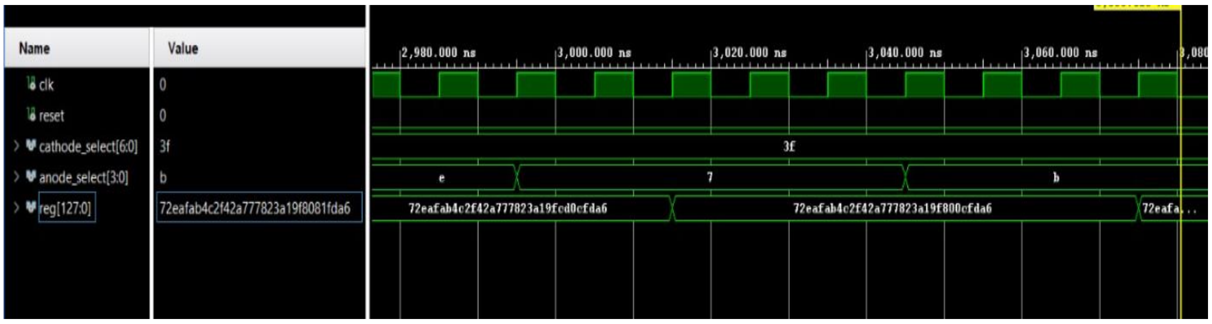
```
Name                  Value
clk                   1
reset                 0
cathode_select[6:0]   3f
anode_select[3:0]     e
reg[127:0]            40872d8dbfe5f52532dbbc26a6cd0cfd

1,900.000 ns    1,920.000 ns    1,940.000 ns    1,960.000 ns
3f
d    e
bd5e7fdfb096b8e9c957204d03c45055    40872d8...
```

ShiftRows:



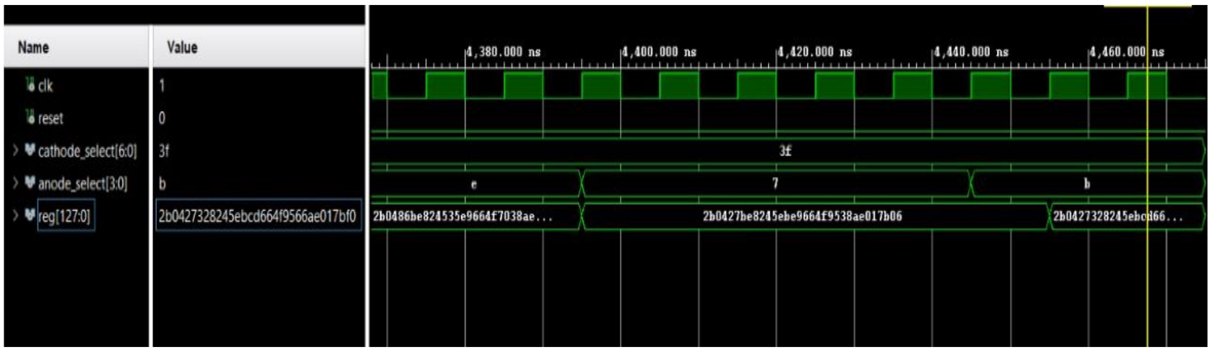
Name	Value
clk	1
reset	0
cathode_select[6:0]	3f
anode_select[3:0]	7
reg[127:0]	40872d8d25bfe5f5bc2632dbcd0cfda6

SubByte:



Name	Value
clk	0
reset	0
cathode_select[6:0]	3f
anode_select[3:0]	b
reg[127:0]	72eafab4c2f42a777823a19f8081fda6

MixCol:



Name	Value
clk	1
reset	0
cathode_select[6:0]	3f
anode_select[3:0]	b
reg[127:0]	2b0427328245ebcd664f9566ae017bf0