COL 215 HARDWARE ASSIGNMENT –3(PART-1)

Himanshi Bhandari(2023CS10888)

Mannat(2023CS10138)

10 November 2024

AES Decryption Implementation Report

This report details the approach and structure of different modules used in implementing AES decryption and their integrated working. The implementation is divided into distinct modules for each primary operation in the AES decryption process. Each module operates on an 8-bit or 32-bit input, processing data based on specified VHDL components that mimic the AES decryption flow. We have created controller/FSM to connect and control each component. Decryption operations are applied in reverse order of encryption, ensuring that the data returns to its original state (plaintext).

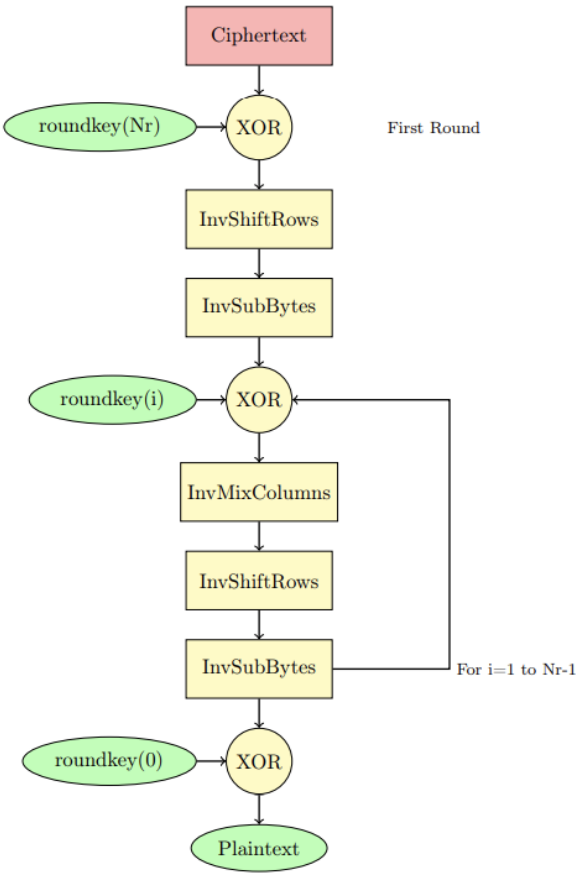


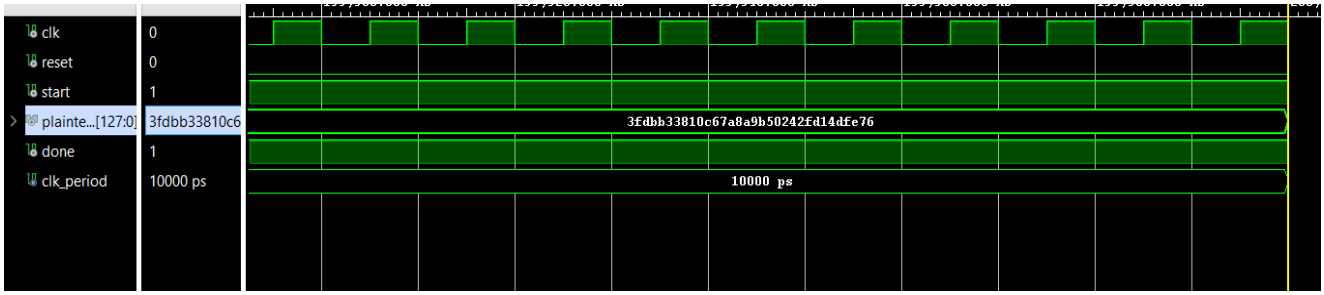
Figure 3: AES Decryption Flowchart

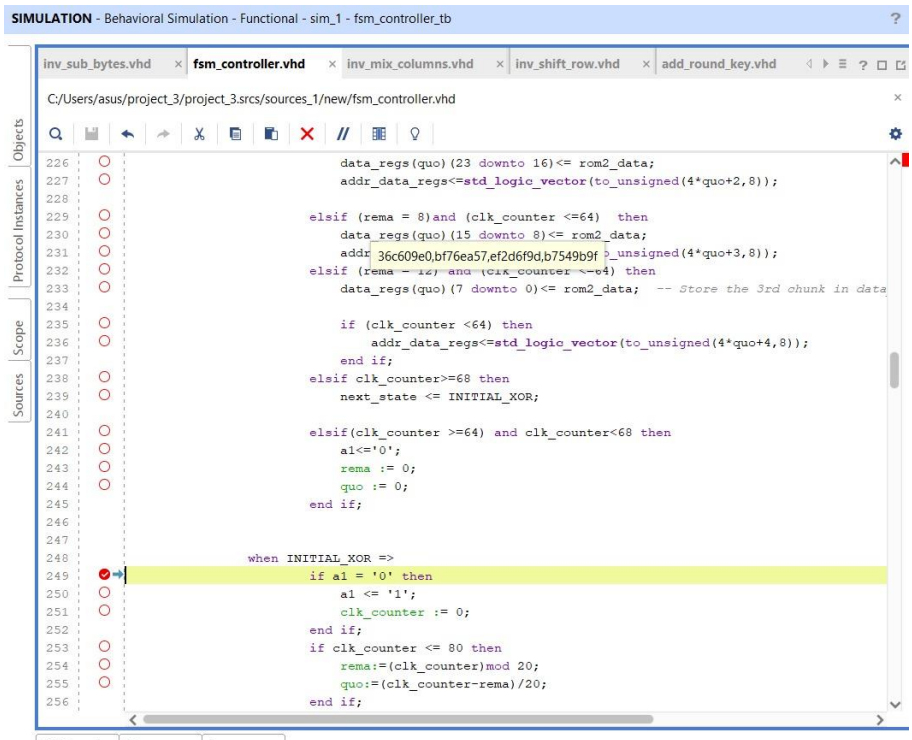
Module Descriptions

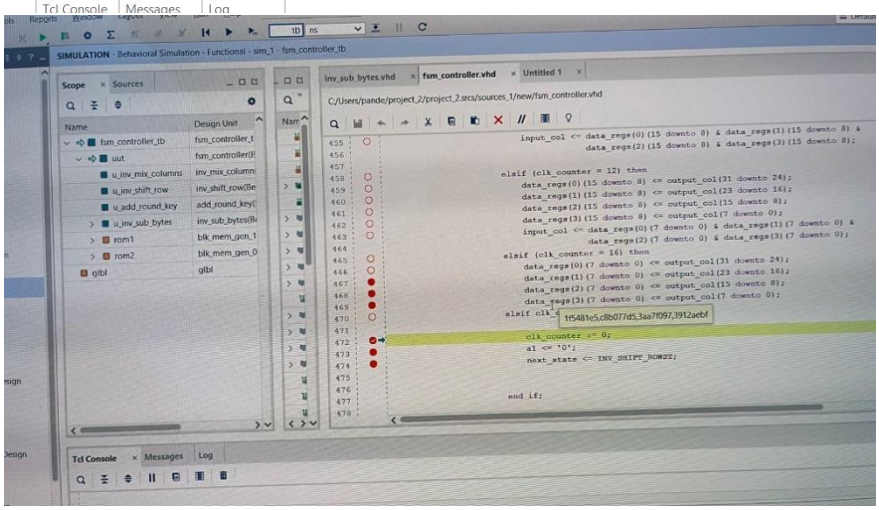
1. FSM MODULE:

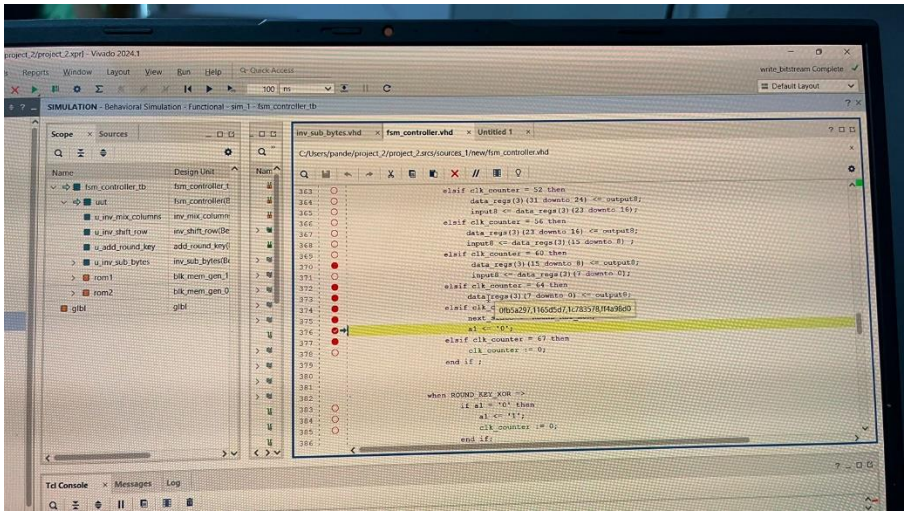
For the control path, we have implemented a common FSM to control both the compute unit and perform data transfers between ROM/RAM and the local registers. The FSM consists of the following states:
IDLE, INIT_READ, INITIAL_XOR, INV_MIX_COLST, INV_SUB_BYTEST, INV_SHIFT_ROWST, INITIAL_INV_SUB_BYTEST, ROUND_KEY_XOR, CHECK_ROUND, FINAL_XOR, INTIAL_INV_SHIFT_ROWST, We have used the next_state signal to change the state and done_sig to know when the decryption is completed. **When done signal become 1 then we display the text using time scrolling.** The FSM consists of two parts: 1. Sequential block: In this block, the state is updated on the clock edge or initialized to a default state if the reset flag is set HIGH. 2. Combinational Block: This consists of the logic to update the next state based on the cur_round values and the current state. The state of the each clock cycle is indicated/controlled in the Combinational block.

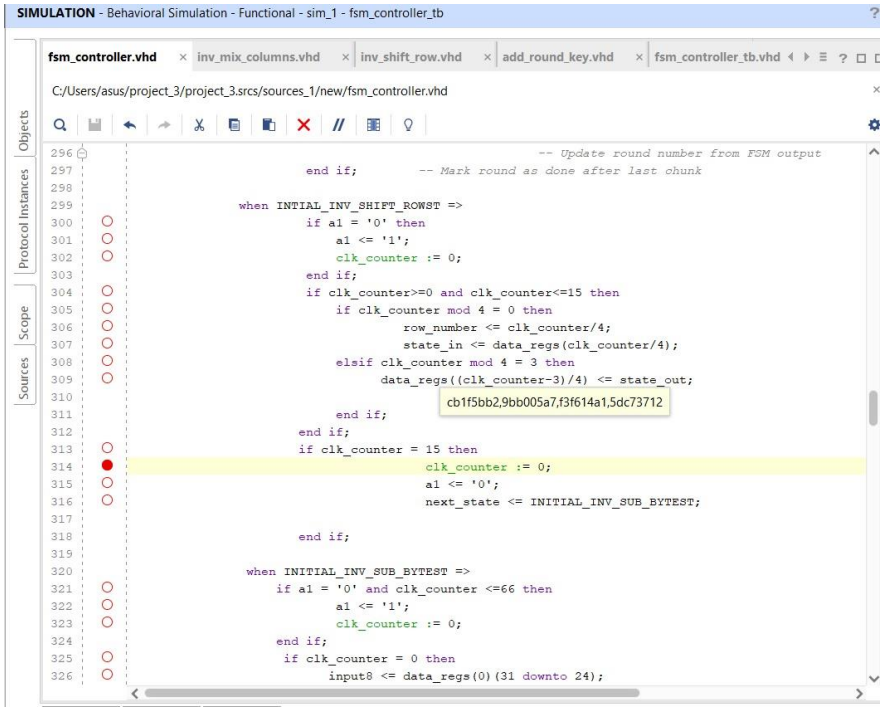
In, Init_read state, we first read the cypher text from ROM and we have created four 32 bits registers data_regs in which we store the rows of the matrix.
Then INITIAL_XOR state is called in which we read the 10^{th} Round keys and store in 4 registers of 32 bits each. And we send the input in chunks of 32 bits to add_round_keys to do xor based on clock cycles.
Once data_regs is updated after getting the output from add_round_keys, we update the cur_round (stores the round number)and the state gets updated to initial__inv_shift_rowst which shifts the rows according to row number which we provide and then data_regs get updated by the output. Here also we send rows as input to inv shift rows module.
After the updation, next state becomes initial_inv_sub_bytest where we have added wait of 4 clock cycles each since reading by rom in inv_sub_byte module will also take time and so we update the registers with the output after this wait. We send 8 bits each in inv_sub_bytes.
Once this operation is completed , state gets updated to round key xor which is similar to initial_xor. After the completion of round_key_xor cur_round gets incremented and state gets updated to inv_mix_colst state. In this state we send columns to the inv mix col module and update the registers by the output which we get. Once it is completed , state gets updated to inv_shift_rowst state which is similar to initial_shift_rowst. After this state becomes inv_sub_bytes which is similar to initial_inv_sub_bytest. Here one iteration of for loop given in flow chart gets completed, now the next state is CHECK_round which check if 8 iterations are completed or not. If they are completed ,then it updates the state to Final xor which similar to previous xor states just that done_sig becomes 1 here that is the decryption is completed and the next state to it is idle. If iterations are left then check round updates the next state to round_key_xor.
We have used 3 Roms, one for inv_s_box, one for cypher text and one for round_keys. We maintain 8 registers of 32 bits to store values(4 for data and 4 for round_keys). And after all the states the value stores in data_regs can be displayed using display.vhd logic. We computed the final decrypted text in fsm_controller and our display.vhd is doing time scrolling properly. We added the logic of display.vhd to our fsm_controller by doing if done_sig is 1 then do the steps which are in process of display.vhd since it will use plaintext_out which we have calculated as a signal .Due to lack of time we could not fix the syntax errors hence submitting the separate files.


clk
0
reset
0
start
1
plainte...[127:0]
3fdbb33810c6
3fdbb33810c67a8a9b50242fd14dfe76
done
1
clk_period
10000 ps
10000 ps


SIMULATION - Behavioral Simulation - Functional - sim_1 - fsm_controller_tb
inv_sub_bytes.vhd
fsm_controller.vhd
inv_mix_columns.vhd
inv_shift_row.vhd
add_round_key.vhd
C:/Users/asus/project_3/project_3.srcs/sources_1/new/fsm_controller.vhd
Objects
Protocol Instances
Scope
Sources
data_regs(quo)(23 downto 16)<= rom2_data;
addr_data_regs<=std_logic_vector(to_unsigned(4*quo+2,8));
elsif (rema = 8)and (clk_counter <=64) then
data_regs(quo)(15 downto 8)<= rom2_data;
36c609e0,bf76ea57,ef2d6f9d,b7549b9f
elsif (rema = 12) and (clk_counter <=64) then
data_regs(quo)(7 downto 0)<= rom2_data; -- Store the 3rd chunk in data
if (clk_counter <64) then
addr_data_regs<=std_logic_vector(to_unsigned(4*quo+4,8));
end if;
elsif clk_counter>=68 then
next_state <= INITIAL_XOR;
elsif(clk_counter >=64) and clk_counter<68 then
a1<='0';
rema := 0;
quo := 0;
end if;
when INITIAL_XOR =>
if a1 = '0' then
a1 <= '1';
clk_counter := 0;
end if;
if clk_counter <= 80 then
rema:=(clk_counter)mod 20;
quo:=(clk_counter-rema)/20;
end if;
Tcl Console
Messages
Log







CHECKING:

We first checked the individual operational modules by creating their testbenches and checking the outputs. They were working correctly. Then we first wrote the code for fsm and then created testbench. We first checked our logic for the code by line by line reading and verifying but we still needed to know the outputs after each step of the state and final output of each state.

FOR CHECKING FINAL OUTPUT OF EACH STATE:

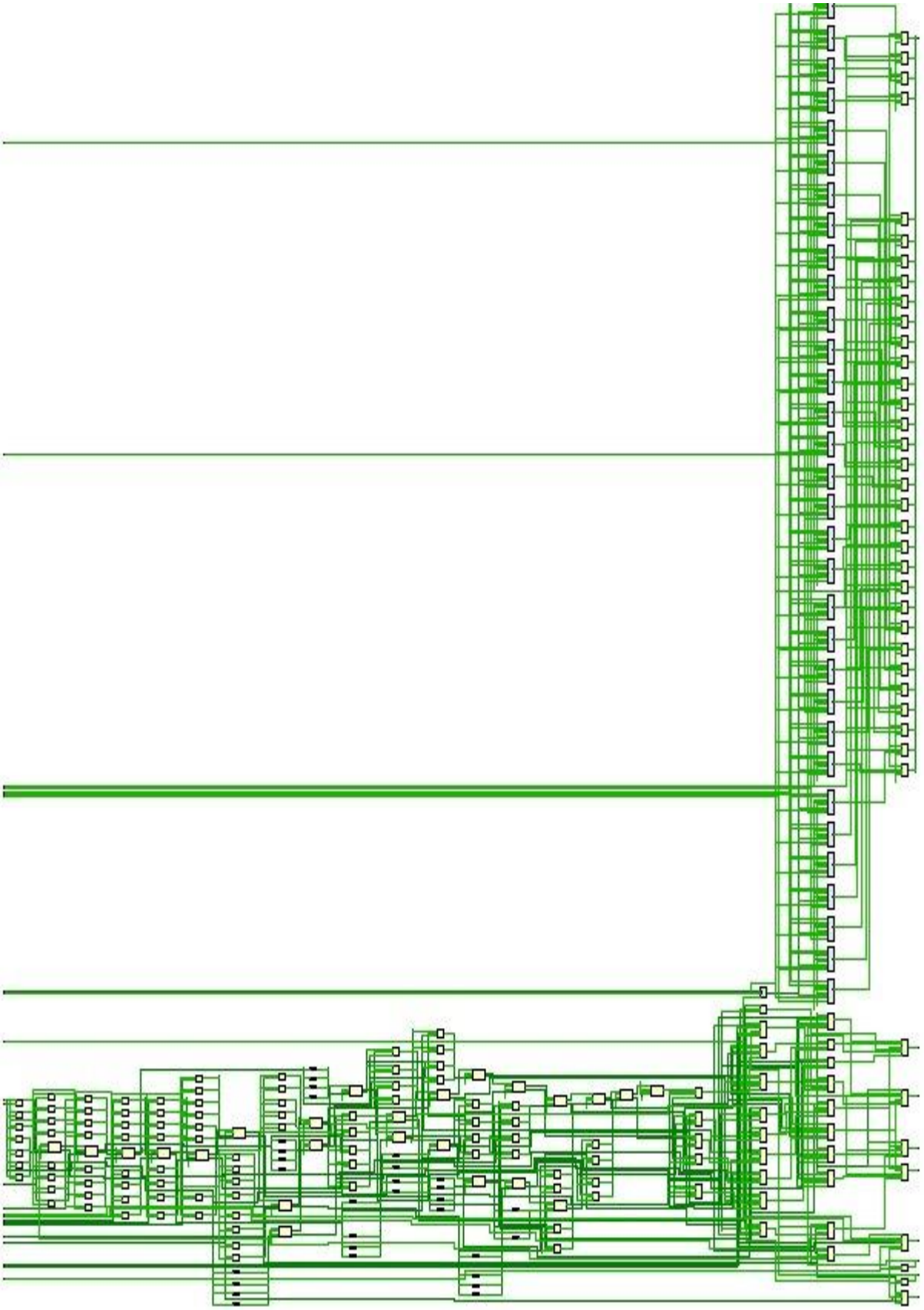
We ran the simulation, and then started by first line of the state and ran simulation for 10ns. Then we hovered above the signal, input, output, register, clk_counter to know the value at the instant. We did this till the state got completed. By checking the value after every 10 ns we were able to debug it thoroughly as we could observe every minute transition in value of registers.

We started by checking INITIAL_READ and we received the correct output after this state.

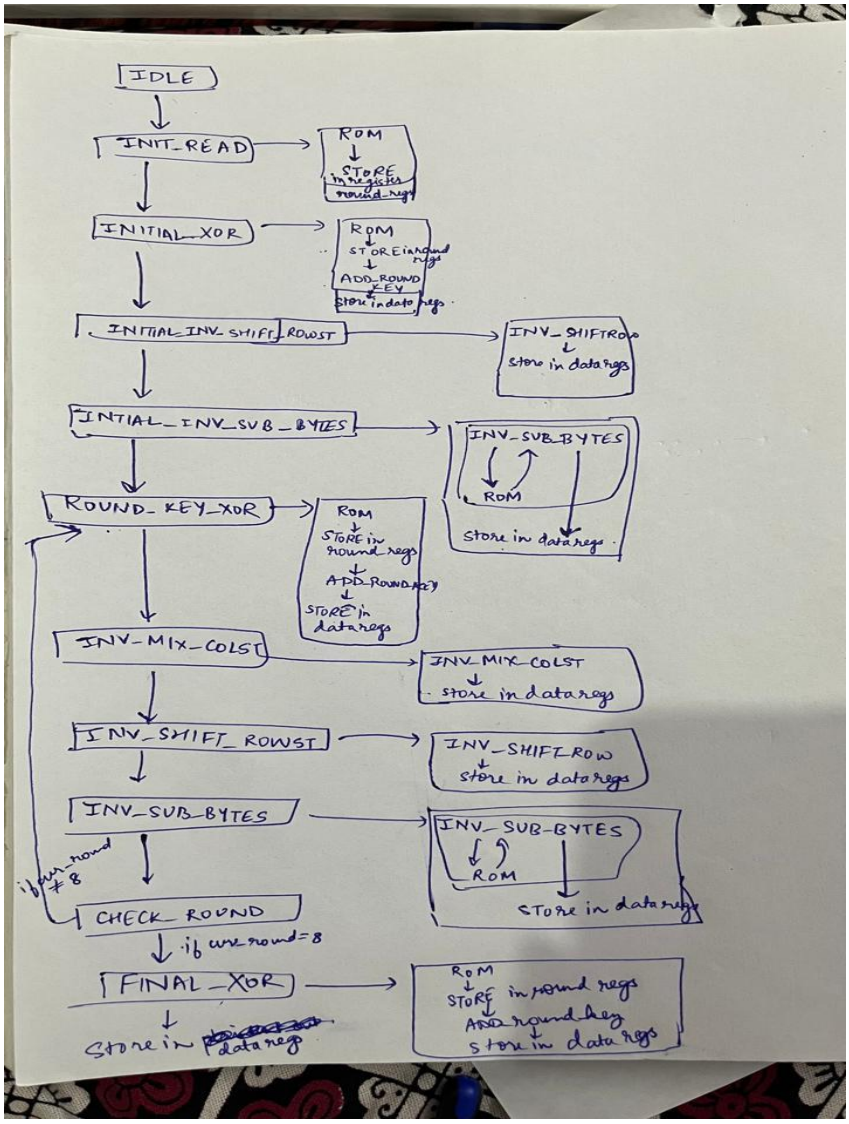
Then before starting the debugging of next state we check if after completion of state it did not repeat the state instead went to the next state, so we ensured this too.

Then we checked the next_state which was INITIAL_XOR for debugging . We knew the value stored in registers. Then we again checked each step and value change after 10-10 ns and also the final output. We verified the output and it was correct. Since the functionality of INITIAL_XOR, ROUND_KEY_XOR and FINAL_XOR is exactly same just one line that is next_state is different, so all these are correct.

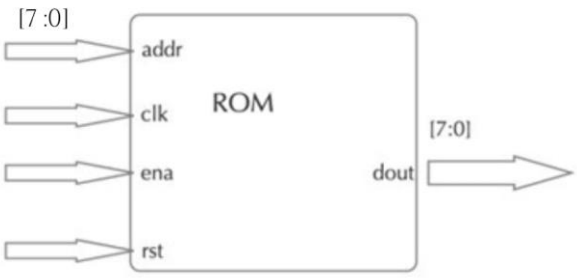
Then we checked for INITIAL_INV_SHIFT_ROWST. We knew the value stored in registers and checked similarly and it was correctly shifting according to row number. Since INV_SHIFT_ROWST is exactly similar except the next_state line hence both are correct.
Then we checked for INITIAL_INV_SUB_BYTEST. We knew the value stored in registers and checked similarly and it was correctly updating the value from inv_s-box. Since INV_SUB_BYTEST is exactly similar except the next_state line hence both are correct.
Then we checked for INV_MIX_COLST and we were getting correct output after this state too.
Also the CHECK_ROUND was correctly working by sending back to round_key_xor when need or else to final_xor.
While debugging we found that initially we had added only one clock cycle but updation of output after computation from operational modules takes time, so we added the time delays accordingly.
I have added a few pictures of our debugging time and how we were getting correct output after each state.

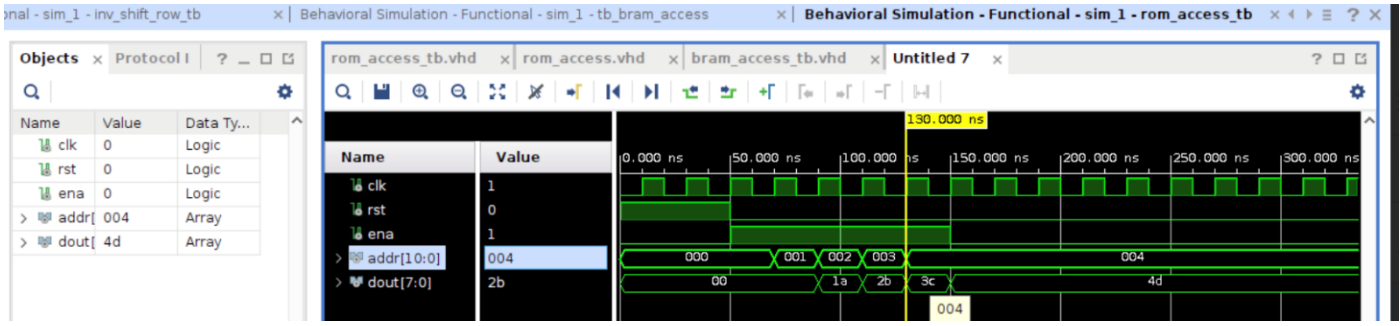


FSM STATE DIAGRAM:

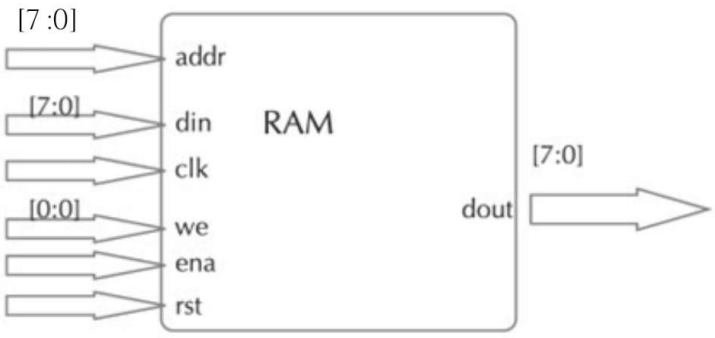


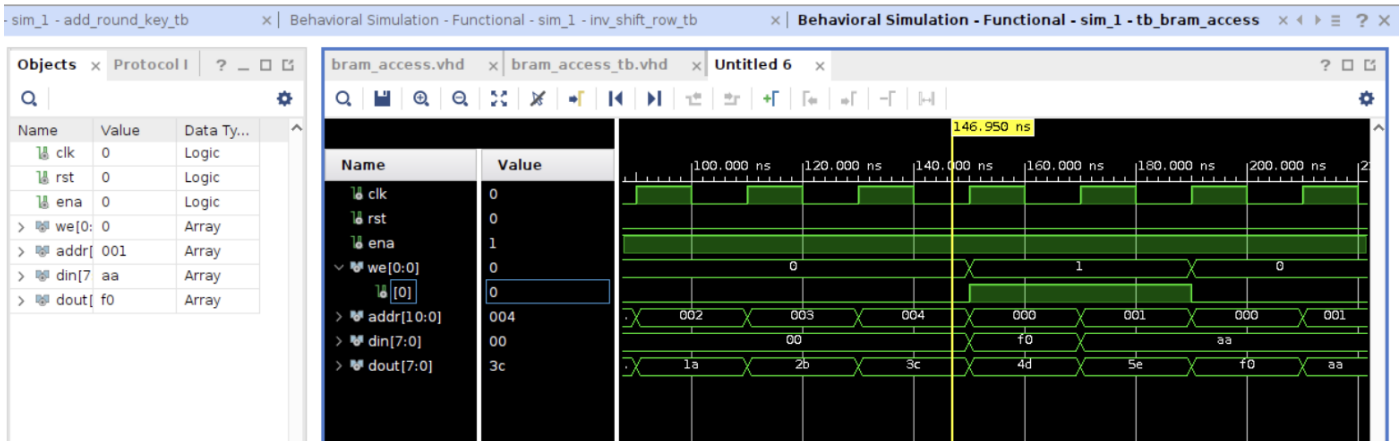
2. **ROM Access Module**: This module loads the cipher text and round keys from a pre-defined .coe file, storing them in ROM to maintain stability across decryption rounds. The ROM interface is controlled by ena, clk, and addr, ensuring synchronous data reads essential for the sequential operations in AES.





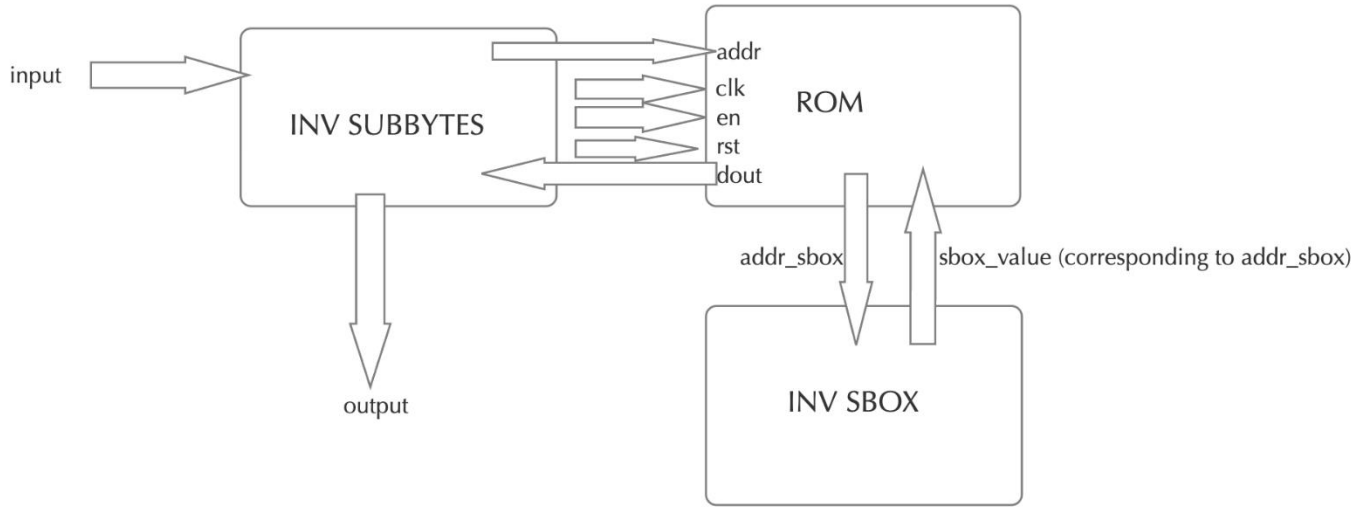
3. **BRAM Access Module**: This module interfaces with a Block RAM (BRAM) component (blk_mem_gen_1), allowing for read and write operations based on control signals such as clock, reset, enable, and write enable. It connects input signals for address and data, facilitating data storage and retrieval in the BRAM.

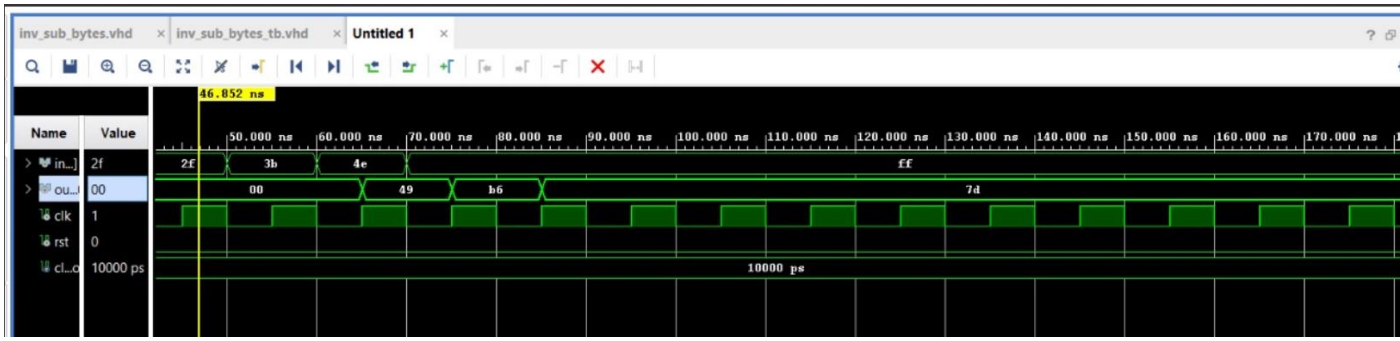
4. 

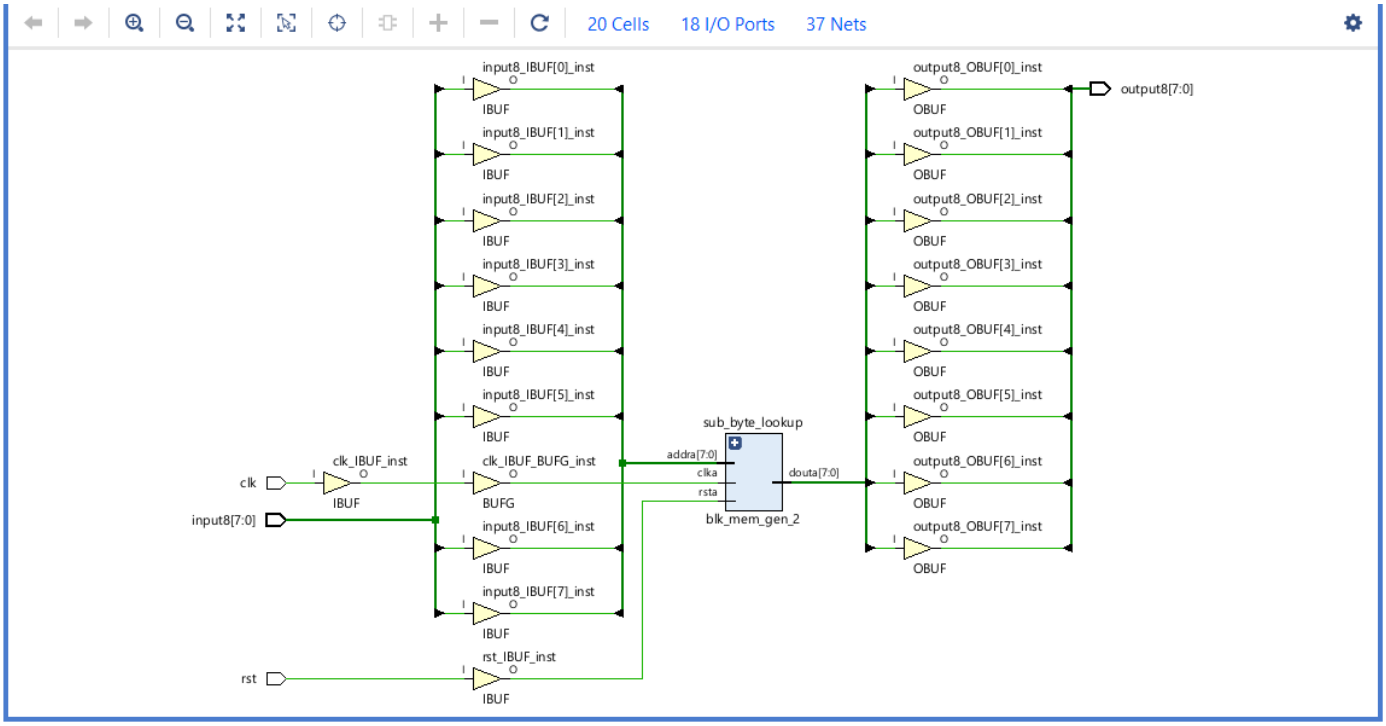


5. **InvSubBytes Module**: The inv_sub_bytes module performs the AES inverse S-Box transformation by referencing a ROM table (blk_mem_gen_2) containing inverse S-Box values. The 8-bit input (input8) serves as an address to retrieve the corresponding inverse S-Box value, outputted as output8, reversing the substitution applied during AES encryption. The module is

synchronized by the clock (clk) and supports reset (rst), which clears the output to zero when active.

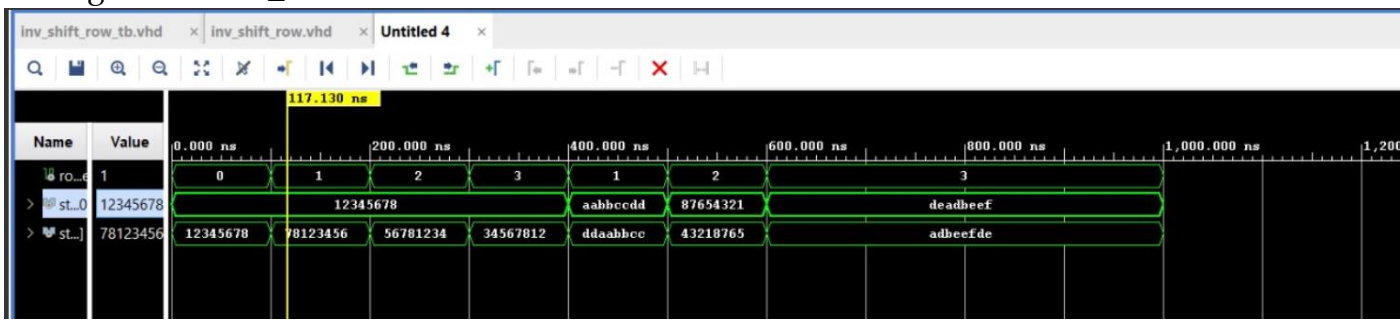


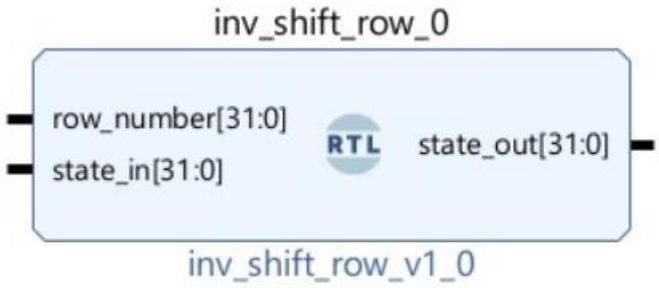


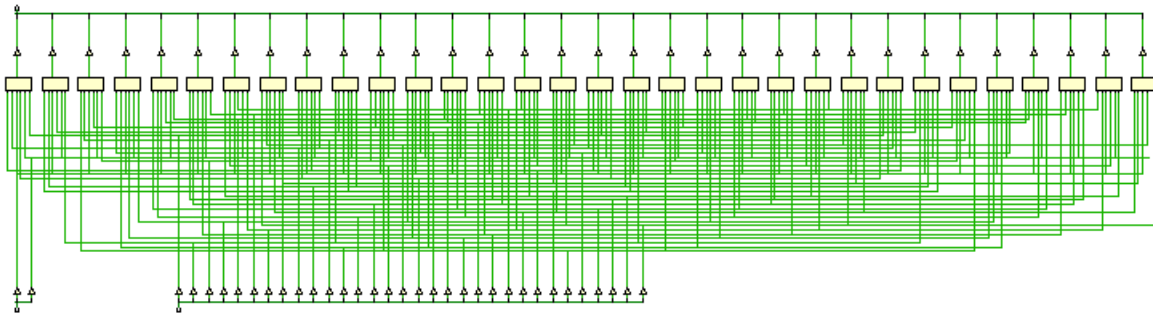


6. **InvShiftRows Module:** This module undoes the row shifts by shifting each row of the state matrix to the right by the required number of bytes, reversing the byte shifts applied during encryption in AES decryption. This operation, essential to restoring the original data arrangement, processes a 128-bit input (state_in) representing a 4x4 matrix of bytes. Each input gets shifted according to the row number and the transformed output

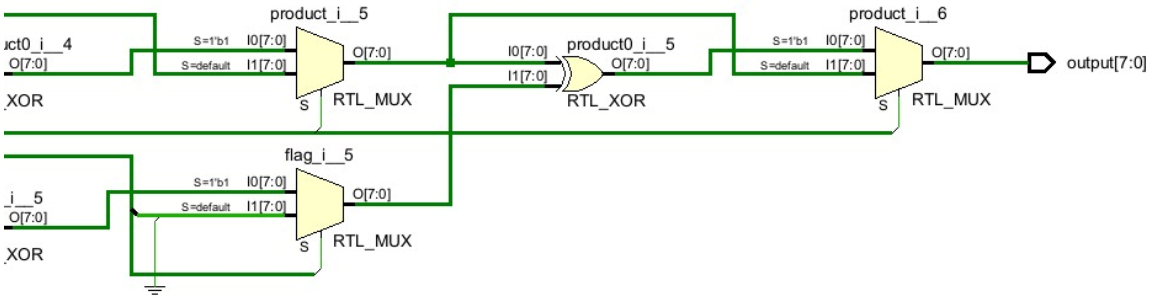
is assigned to state_out

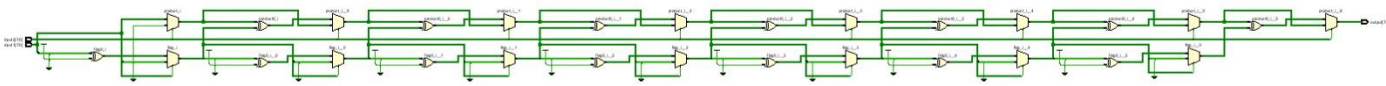


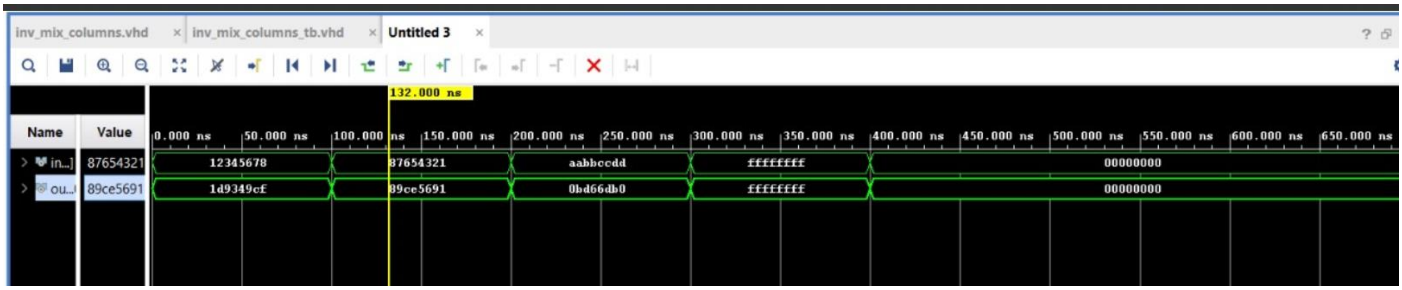




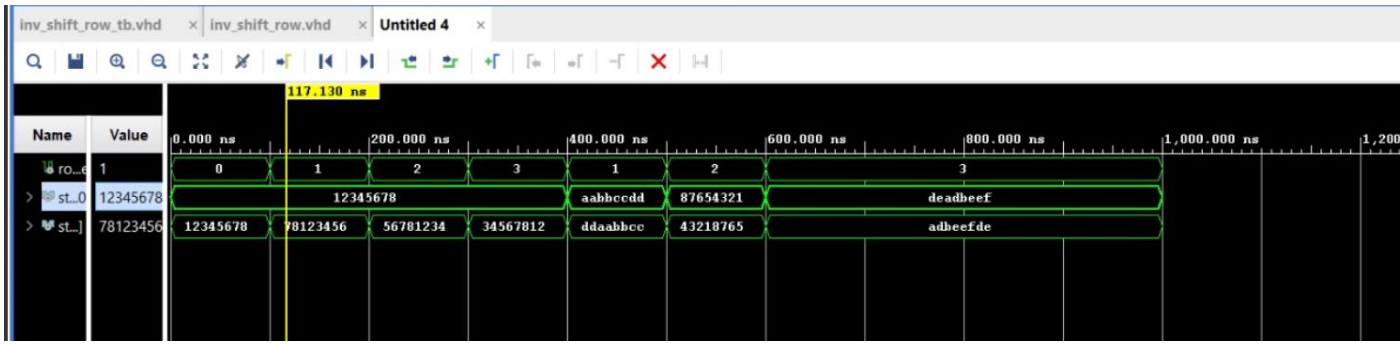
7. **Inv_mix_columns:** The inv_mix_columns module implements the inverse MixColumns transformation in AES decryption, using Galois Field (GF(2^8)) multiplication to reverse the column mixing applied during encryption. The module accepts a 32-bit input (input_col), which represents a column of four 8-bit values, and outputs a 32-bit transformed column (output_col). Inside the module, each byte of input_col is split into col_0 through col_3. The function multiplier_of_two performs GF(2^8) multiplication using bitwise shifts and XOR operations with the irreducible polynomial 0x1B for polynomial reduction, ensuring results remain within GF(2^8). The inverse MixColumns transformation is computed by multiplying each column byte with constants (0E, 0B, 0D, 09), arranged to reverse encryption's mixing step. Each resulting row (row_0 through row_3) is computed by XORing the GF-multiplication outputs, and these rows are concatenated to form output_col, effectively transforming the input column for AES decryption.

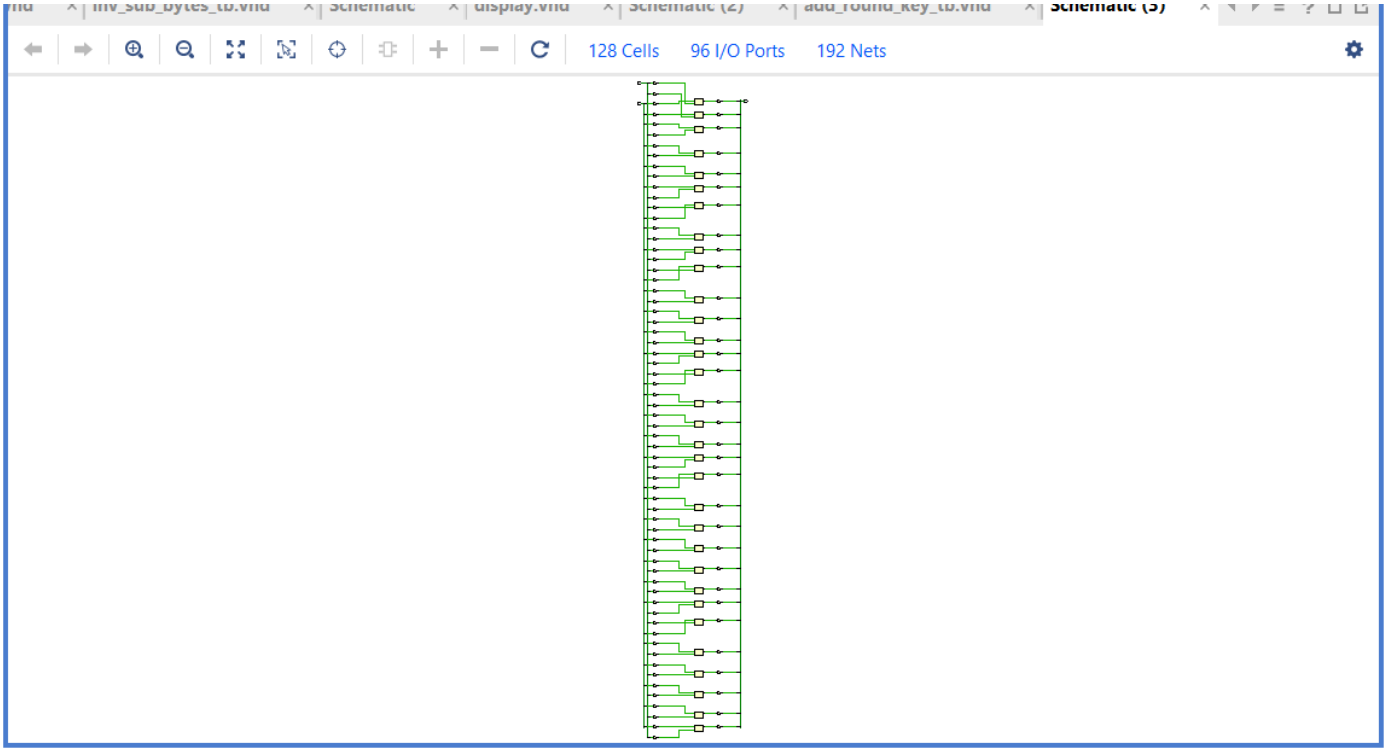




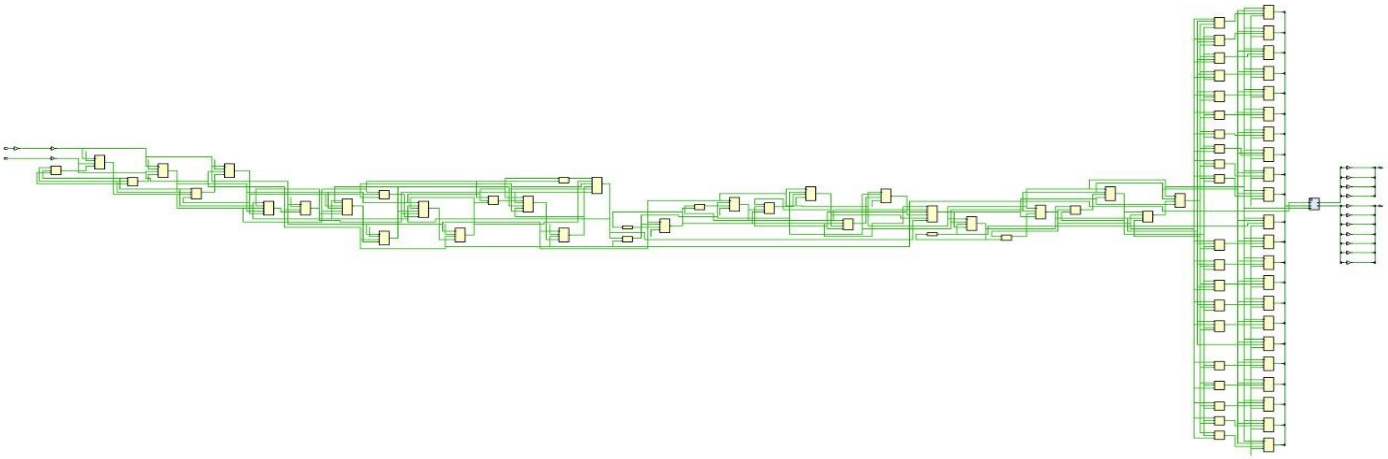


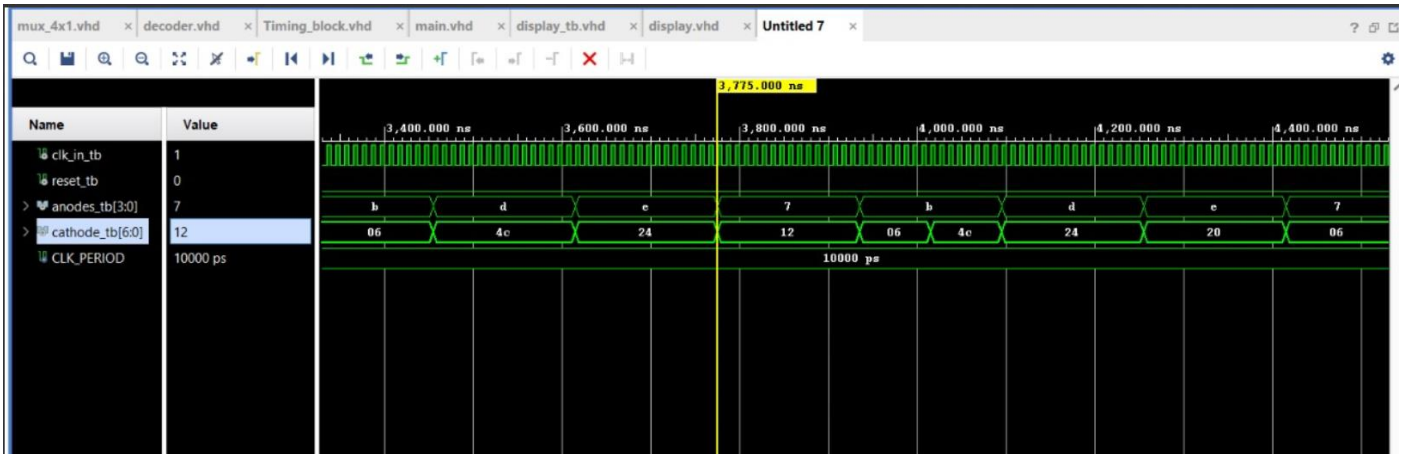
8. **AddRoundKey Module:** The add_round_key module implements the AddRoundKey step in AES by performing a bitwise XOR operation between two 32-bit inputs (input1 and input2), which represent a portion of the state and the round key, respectively. Each 8-bit segment of input1 is XORed with the corresponding 8-bit segment of input2, producing a 32-bit output (output) that represents the transformed state. This operation combines the round key with the current state, a key step in AES encryption and decryption, effectively scrambling the data based on the key for security.

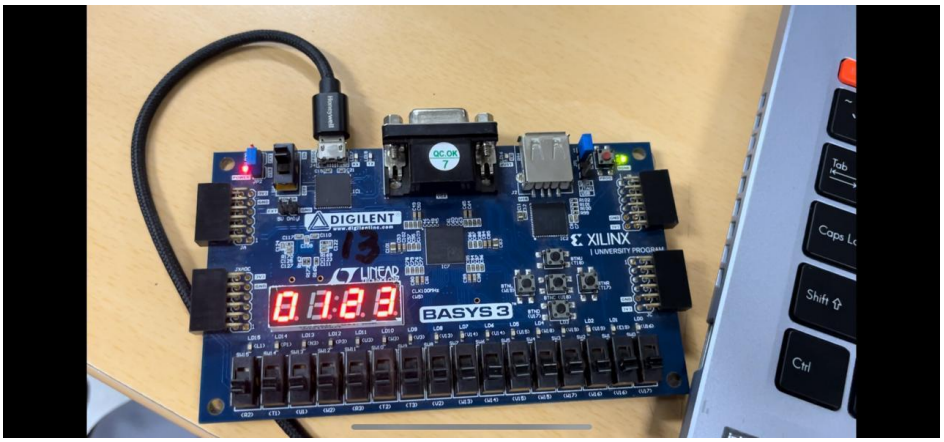


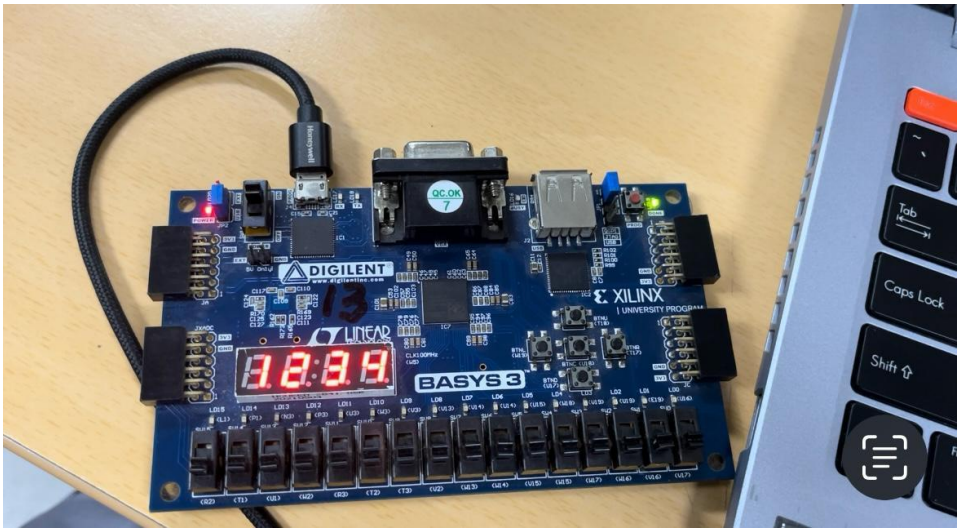


9. **Display module:** The display module controls a scrolling 4-digit display on a Basys 3 board, driven by a 100 MHz clock (clk_in) and reset signal. A 128-bit data input (data_in) is displayed sequentially by sending four 8-bit segments to the main component, which manages the timing, decoding, and multiplexing for each digit on the 7-segment display. N is a constant that sets the clock divider rate, controlling the speed of the scrolling display. To control the scroll speed, a clock divider (clk_divider) slows down the update rate, while counters (scroll_count and shift_position) track and shift through the data segments in 8-bit (1-byte) increments. The process loops through all 12 segments in data_in and then resets, continuously displaying each 4-byte portion. The main component refreshes the displays at a rate fast enough to create a smooth scrolling effect. This VHDL module uses the decoder which decodes an 8-bit binary input into a 7-segment display output for hexadecimal characters (0-9, A-F) and space, using active-low logic. The process block's case statement matches the input to specific ASCII values, turning on segments (set to '0') to form each character. Unmatched inputs result in a blank (-) display. Also, we have replaced the decoder of HW 2 with this decoder. So logic for timing block remains similar to previous time.




mux_4x1.vhd
decoder.vhd
Timing_block.vhd
main.vhd
display_tb.vhd
display.vhd
Untitled 7
Name
Value
clk_in_tb
1
reset_tb
0
anodes_tb[3:0]
7
cathode_tb[6:0]
12
CLK_PERIOD
10000 ps
3,775.000 ns
3,400.000 ns
3,600.000 ns
3,800.000 ns
4,000.000 ns
4,200.000 ns
4,400.000 ns
b d e 7 b d e 7
06 4c 24 12 06 4c 24 20 06
10000 ps


DIGILENT
XILINX
BASYS 3
Tab
Caps Lo
Shift
Ctrl



We have added the testcases in tb files.

Conclusion

We have implemented AES Decryption and created controller/FSM to connect and control each component.