Neil Chulani

Sequential Implementation of AES-128 on FPGA

Introduction

The main objective for this project was to understand the specifications of AES-128 and implement the encryption (and key scheduling) algorithm on the FPGA. The purpose of this is to use the FPGA as a hardware accelerator to increase the speed of the encryption process. Performing AES encryption is much easier to do programmatically on the MCU, but would take much longer to do, making a hardware implementation viable. One big focus point of the lab was the limitations of our FPGA; due to the size of the FPGA we are using, we cannot just implement all of the rounds in one large block of combinational logic. We had to design a sequential process to fit the architecture on the FPGA, which will be described in detail in the following sections.

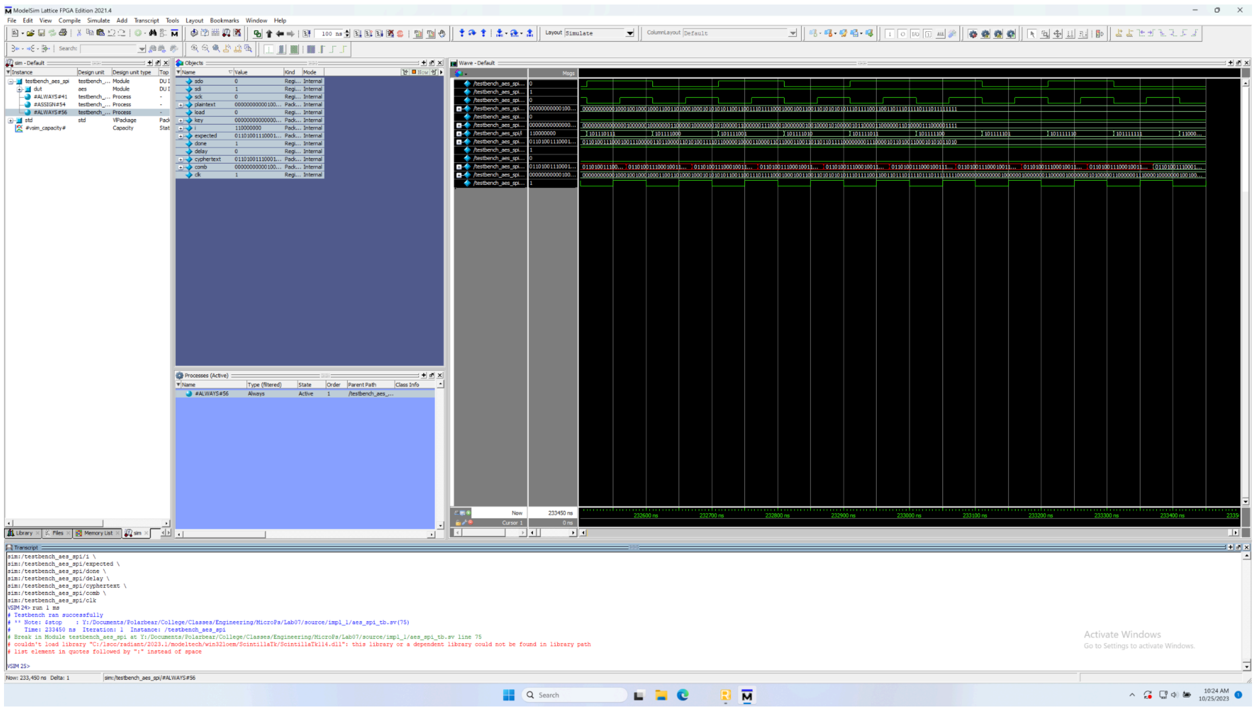
Design and Testing Methodology

To start this design, I created the modules and testbenches for each of the “helper” modules. These included Add Round Key, Rot Word, Shift Rows, Sub Bytes, and Sub Word. These helper modules are used during key expansion and during each round. By designing and testing each module separately, I could go into the process of creating logic to perform the overall encryption with a strong foundation and confidence that each separate module functions correctly.

The main architecture of my implementation is based on a large FSM that performs both the key expansion and the encryption. By using an FSM, I can ensure that the key is correctly expanded for the round before beginning any encryption operations for that round. The FSM keeps track of the round and continuously updates the expanded key and the ciphertext until it finishes round 10, at which point the encryption process is finished and it can mark the done flag. I designed a testbench to test this FSM based on the provided AES test vectors in the specifications, and then used the FSM in the aes_core module. The aes_code module itself is a small FSM that helps control when to encrypt a new plaintext message with a new key. One thing that I had to change when going from the FSM to the aes_core module was that I had to add a delay at the beginning to ensure that the aes_core module had time to properly populate the plaintext and the key. I had to add a larger delay when testing the aes_spi module for a similar reason, to give the module time to populate the key and plaintext correctly. After adding the delays before the encryption process in the FSM module, both the aes_core and the aes_spi testbenches ran successfully.

Technical Documentation

System Verilog Code for AES_SPI was not to be modified.
AES_SPI Testbench Output (Alternate Testcase):

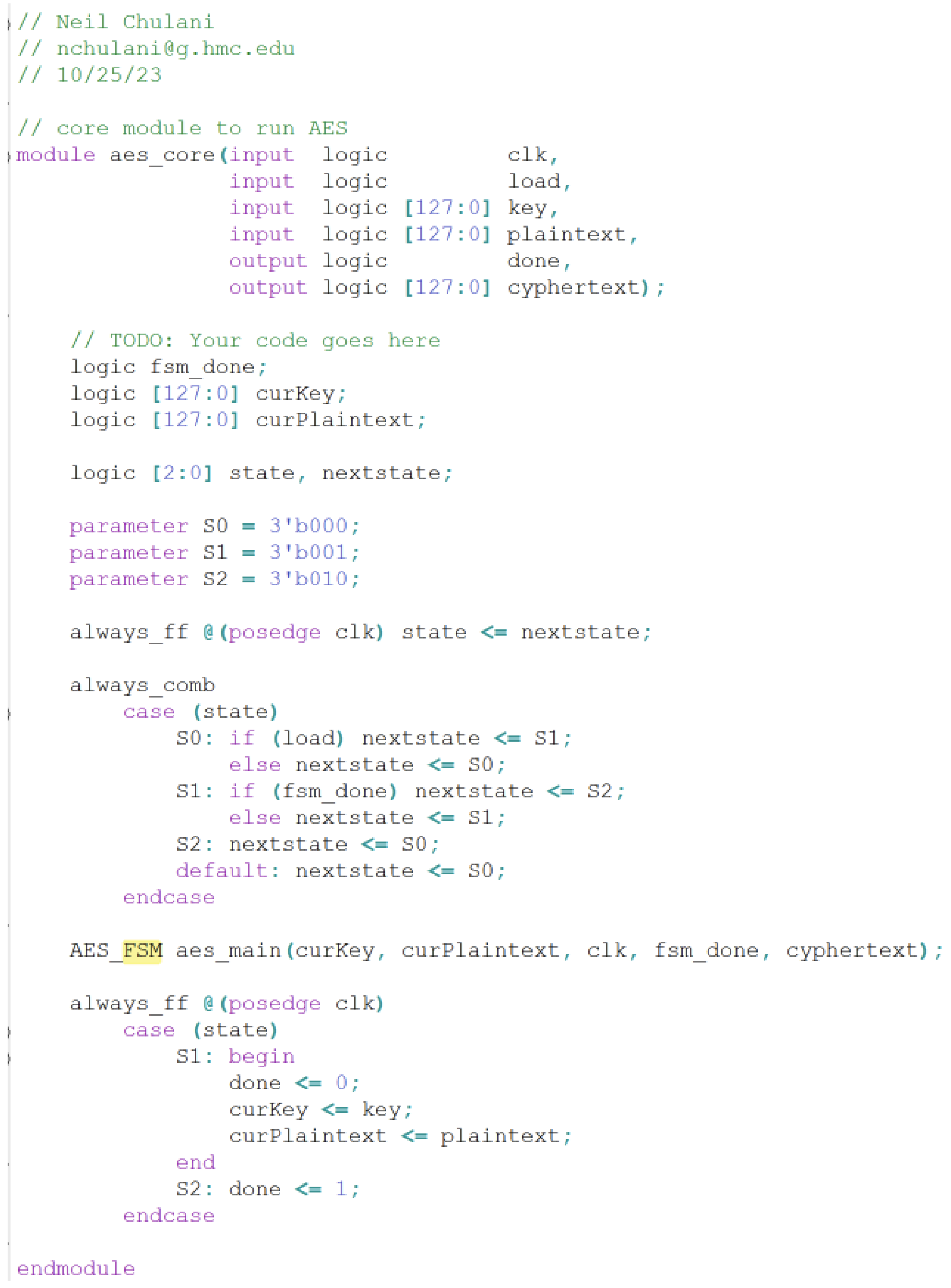


```
sim:/testbench_aes_spi/i \
sim:/testbench_aes_spi/expected \
sim:/testbench_aes_spi/done \
sim:/testbench_aes_spi/delay \
sim:/testbench_aes_spi/cyphertext \
sim:/testbench_aes_spi/comb \
sim:/testbench_aes_spi/clk
VSIM 24> run 1 ms
# Testbench ran successfully
# ** Note: $stop    : Y:/Documents/Polarbear/College/Classes/Engineering/MicroPs/Lab07/source/impl_1/aes_spi_tb.sv(75)
#    Time: 233450 ns  Iteration: 1  Instance: /testbench_aes_spi
# Break in Module testbench_aes_spi at Y:/Documents/Polarbear/College/Classes/Engineering/MicroPs/Lab07/source/impl_1/aes_spi_tb.sv line 75
# couldn't load library "C:/lscc/radiant/2023.1/modeltech/win32loem/ScintillaTk/ScintillaTk114.dll": this library or a dependent library could not be found in library path
# list element in quotes followed by ":" instead of space

VSIM 25>
```

Now: 233,450 ns Delta: 1   sim:/testbench_aes_spi/#ALWAYS#56

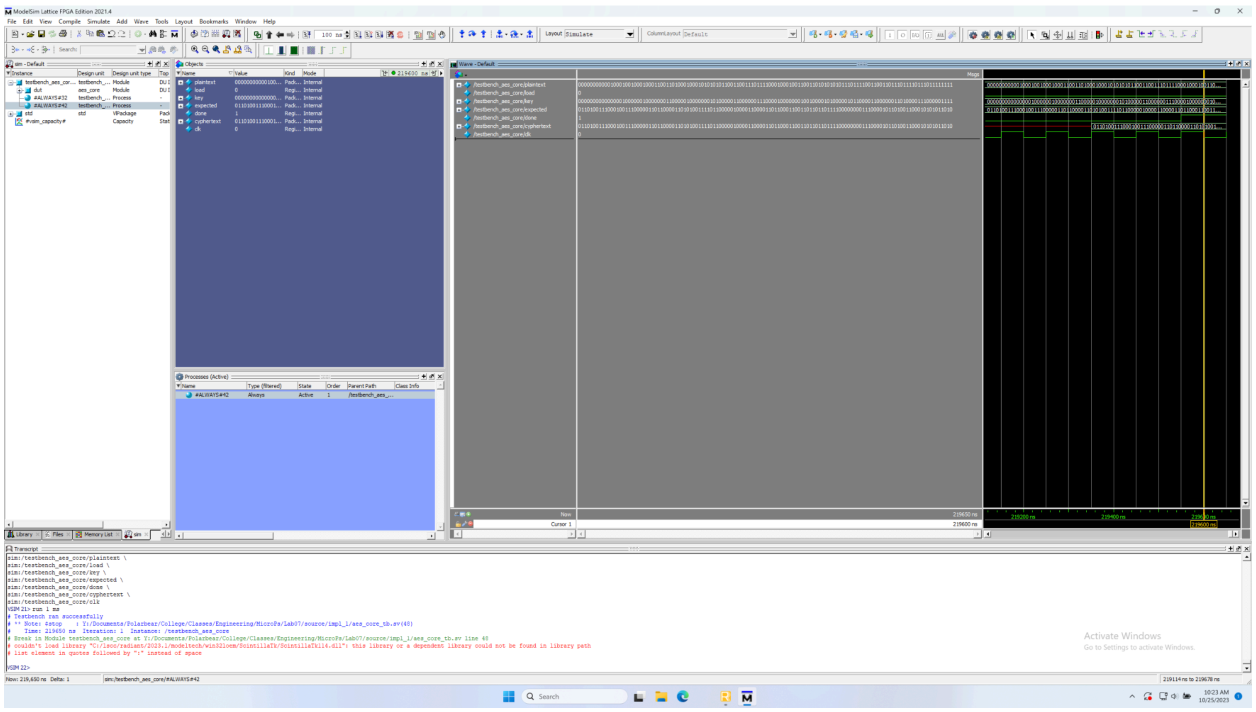
Note: the error is just a path error I have on Parallels Desktop VM, but we can see the testbench successful message before it

SystemVerilog Code for AES_CORE :



```
// Neil Chulani
// nchulani@g.hmc.edu
// 10/25/23

// core module to run AES
module aes_core(input  logic         clk,
                input  logic         load,
                input  logic [127:0] key,
                input  logic [127:0] plaintext,
                output logic         done,
                output logic [127:0] cyphertext);

    // TODO: Your code goes here
    logic fsm_done;
    logic [127:0] curKey;
    logic [127:0] curPlaintext;

    logic [2:0] state, nextstate;

    parameter S0 = 3'b000;
    parameter S1 = 3'b001;
    parameter S2 = 3'b010;

    always_ff @(posedge clk) state <= nextstate;

    always_comb
        case (state)
            S0: if (load) nextstate <= S1;
                else nextstate <= S0;
            S1: if (fsm_done) nextstate <= S2;
                else nextstate <= S1;
            S2: nextstate <= S0;
            default: nextstate <= S0;
        endcase

    AES_FSM aes_main(curKey, curPlaintext, clk, fsm_done, cyphertext);

    always_ff @(posedge clk)
        case (state)
            S1: begin
                done <= 0;
                curKey <= key;
                curPlaintext <= plaintext;
            end
            S2: done <= 1;
        endcase

endmodule
```

AES_CORE Testbench Output (Alternate Testcase):



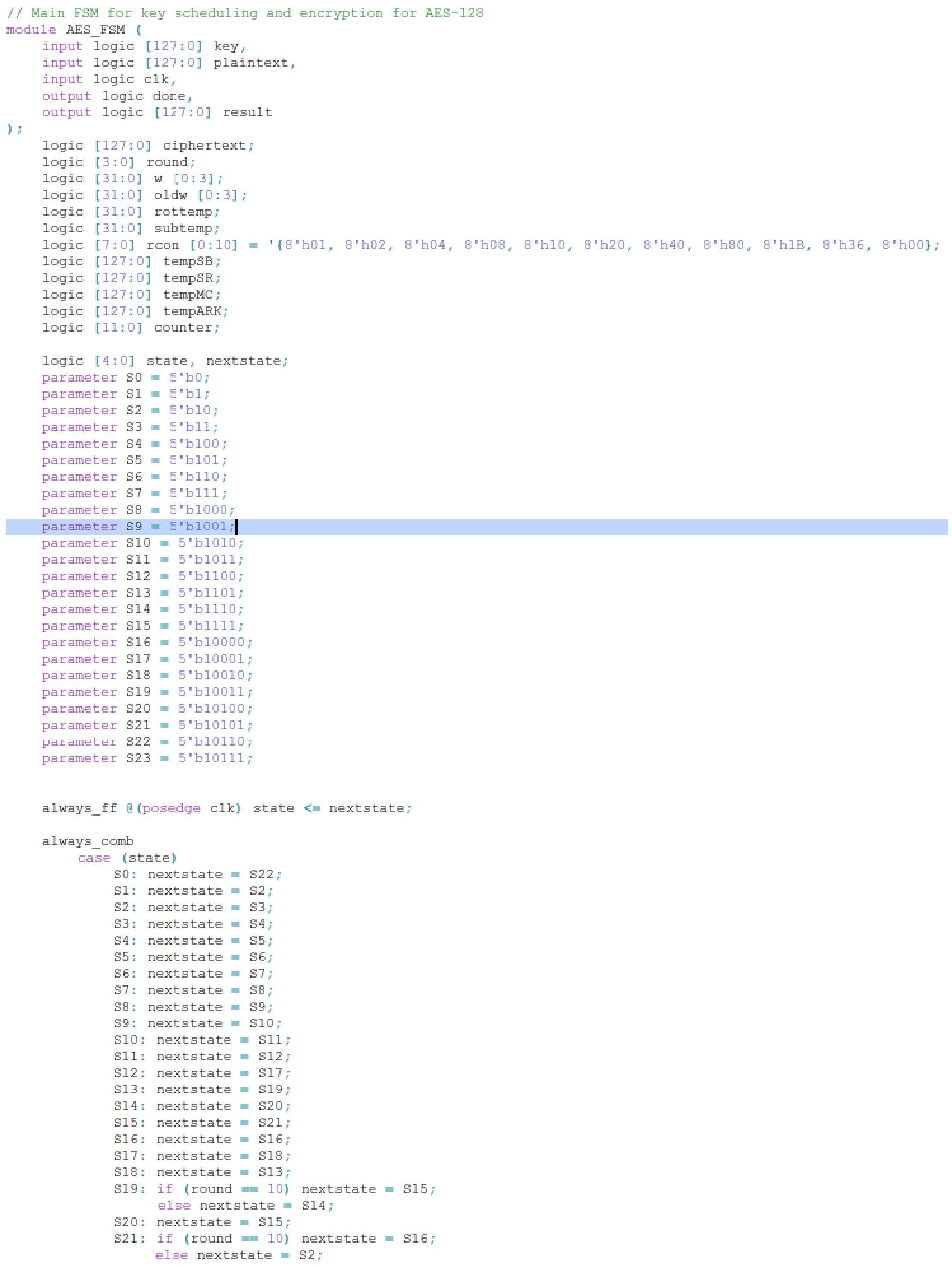
Name	Value	Kind	Mode
plaintext	00000000000100...	Pack...	Internal
load	0	Regi...	Internal
key	0000000000000...	Pack...	Internal
expected	01101001110001...	Pack...	Internal
done	1	Regi...	Internal
cyphertext	01101001110001...	Pack...	Internal
clk	0	Regi...	Internal

```
sim:/testbench_aes_core/plaintext \
sim:/testbench_aes_core/load \
sim:/testbench_aes_core/key \
sim:/testbench_aes_core/expected \
sim:/testbench_aes_core/done \
sim:/testbench_aes_core/cyphertext \
sim:/testbench_aes_core/clk
VSIM 21> run 1 ms
# Testbench ran successfully
# ** Note: $stop    : Y:/Documents/Polarbear/College/Classes/Engineering/MicroPs/Lab07/source/impl_1/aes_core_tb.sv(48)
#    Time: 219650 ns  Iteration: 1  Instance: /testbench_aes_core
# Break in Module testbench_aes_core at Y:/Documents/Polarbear/College/Classes/Engineering/MicroPs/Lab07/source/impl_1/aes_core_tb.sv line 48
# couldn't load library "C:/lscc/radiant/2023.1/modeltech/win32loem/ScintillaTk/ScintillaTk114.dll": this library or a dependent library could not be found in library path
# list element in quotes followed by ":" instead of space

VSIM 22>
```

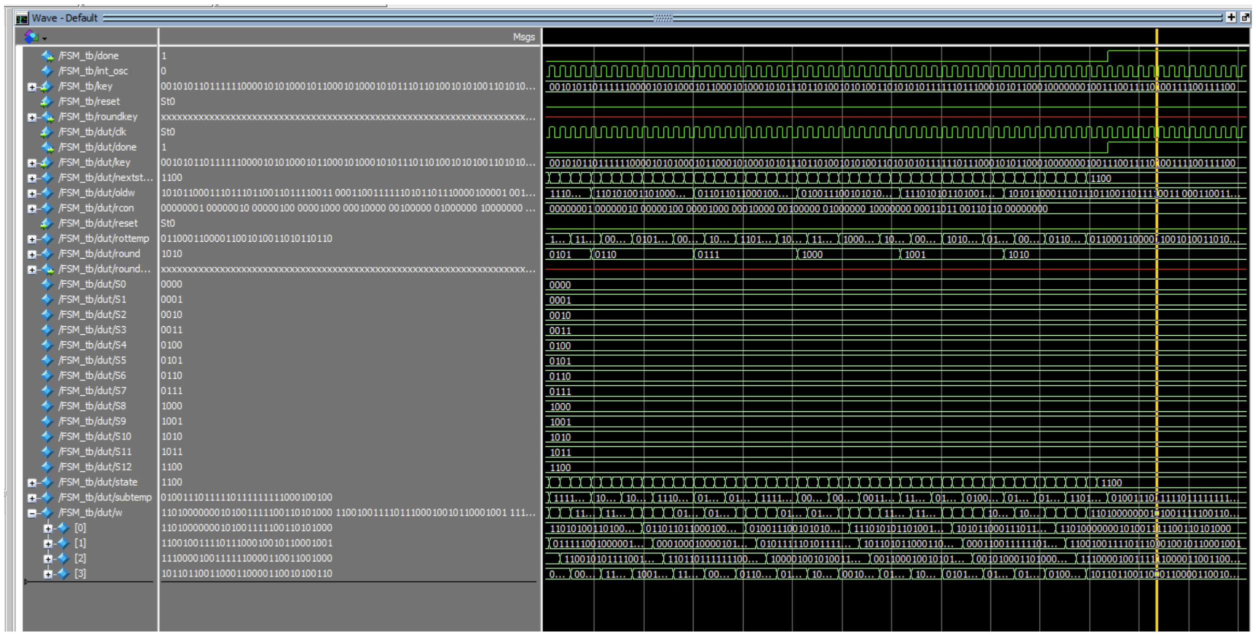
Now: 219,650 ns Delta: 1   sim:/testbench_aes_core/#ALWAYS#42

Note: the error is just a path error I have on Parallels Desktop VM, but we can see the testbench successful message before it

SystemVerilog Code for AES_FSM:



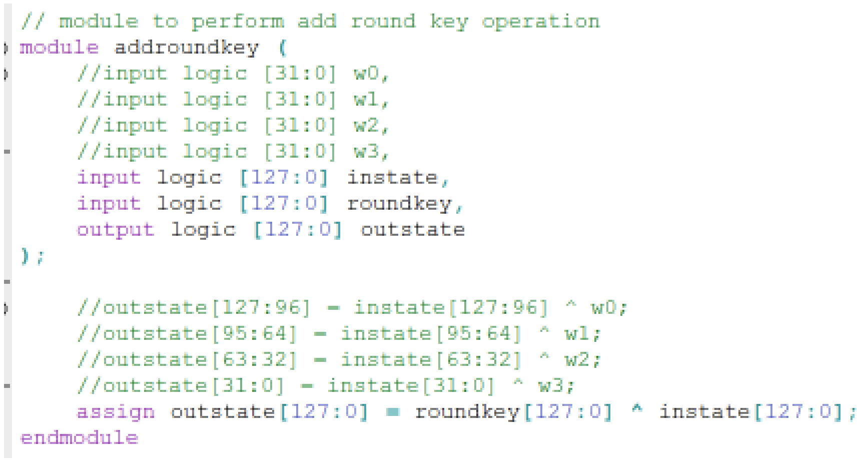
```
// Main FSM for key scheduling and encryption for AES-128
module AES_FSM (
    input logic [127:0] key,
    input logic [127:0] plaintext,
    input logic clk,
    output logic done,
    output logic [127:0] result
);
    logic [127:0] ciphertext;
    logic [3:0] round;
    logic [31:0] w [0:3];
    logic [31:0] oldw [0:3];
    logic [31:0] rottemp;
    logic [31:0] subtemp;
    logic [7:0] rcon [0:10] = '{8'h01, 8'h02, 8'h04, 8'h08, 8'h10, 8'h20, 8'h40, 8'h80, 8'h1B, 8'h36, 8'h00};
    logic [127:0] tempSB;
    logic [127:0] tempSR;
    logic [127:0] tempMC;
    logic [127:0] tempARK;
    logic [11:0] counter;

    logic [4:0] state, nextstate;
    parameter S0 = 5'b0;
    parameter S1 = 5'b1;
    parameter S2 = 5'b10;
    parameter S3 = 5'b11;
    parameter S4 = 5'b100;
    parameter S5 = 5'b101;
    parameter S6 = 5'b110;
    parameter S7 = 5'b111;
    parameter S8 = 5'b1000;
    parameter S9 = 5'b1001;
    parameter S10 = 5'b1010;
    parameter S11 = 5'b1011;
    parameter S12 = 5'b1100;
    parameter S13 = 5'b1101;
    parameter S14 = 5'b1110;
    parameter S15 = 5'b1111;
    parameter S16 = 5'b10000;
    parameter S17 = 5'b10001;
    parameter S18 = 5'b10010;
    parameter S19 = 5'b10011;
    parameter S20 = 5'b10100;
    parameter S21 = 5'b10101;
    parameter S22 = 5'b10110;
    parameter S23 = 5'b10111;


    always_ff @(posedge clk) state <= nextstate;

    always_comb
        case (state)
            S0: nextstate = S22;
            S1: nextstate = S2;
            S2: nextstate = S3;
            S3: nextstate = S4;
            S4: nextstate = S5;
            S5: nextstate = S6;
            S6: nextstate = S7;
            S7: nextstate = S8;
            S8: nextstate = S9;
            S9: nextstate = S10;
            S10: nextstate = S11;
            S11: nextstate = S12;
            S12: nextstate = S17;
            S13: nextstate = S19;
            S14: nextstate = S20;
            S15: nextstate = S21;
            S16: nextstate = S16;
            S17: nextstate = S18;
            S18: nextstate = S13;
            S19: if (round == 10) nextstate = S15;
                 else nextstate = S14;
            S20: nextstate = S15;
            S21: if (round == 10) nextstate = S16;
                 else nextstate = S2;
            S22: if (counter == 2000) nextstate = S23;
                 else nextstate = S22;
            S23: nextstate = S1;
            default: nextstate = S0;
        endcase

    rotword rot ('{w[3][31:24], w[3][23:16], w[3][15:8], w[3][7:0]}, '{rottemp[31:24], rottemp[23:16], rottemp[15:8], rottemp[7:0]});
    subword sub ('{w[3][31:24], w[3][23:16], w[3][15:8], w[3][7:0]}, clk, '{subtemp[31:24], subtemp[23:16], subtemp[15:8], subtemp[7:0]});
    addroundkey ark(ciphertext, {w[0], w[1], w[2], w[3]}, tempARK);
    subbytes sb(ciphertext, clk, tempSB);
    shiftrows sr(ciphertext, tempSR);
    mixcolumns mc(ciphertext, tempMC);

    always_ff @(posedge clk)
        case (state)
            S0: counter <= 0;
            S22: counter <= counter + 1;
            S23: begin
                done <= 0;
                round <= 0;
                w[0] <= key[127:96];
                w[1] <= key[95:64];
                w[2] <= key[63:32];
                w[3] <= key[31:0];
                ciphertext <= plaintext;
            end
            S1: ciphertext <= tempARK;
            S2: begin
                round <= round + 1;
                oldw[0] <= w[0];
                oldw[1] <= w[1];
                oldw[2] <= w[2];
                oldw[3] <= w[3];
            end
            S3: w[3] <= rottemp;
            S6: w[3] <= subtemp;
            S7: w[0] <= oldw[0] ^ w[3] ^ (rcon[round - 1] << 24);
            S8: w[1] <= oldw[1] ^ w[0];
            S9: w[2] <= oldw[2] ^ w[1];
            S10: w[3] <= oldw[3] ^ w[2];
            S11: ciphertext <= tempSB;
            S13: ciphertext <= tempSR;
            S14: ciphertext <= tempMC;
            S15: ciphertext <= tempARK;
            S16: begin
                done <= 1;
                result <= ciphertext;
            end
        endcase
endmodule
```



```
Wave - Default
                                   Msgs
/FSM_tb/done            1
/FSM_tb/int_osc         0
/FSM_tb/key             00101011011111100001010100010110001010001010111011010010101001101010...
/FSM_tb/reset           St0
/FSM_tb/roundkey        xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx...
/FSM_tb/dut/clk         St0
/FSM_tb/dut/done        1
/FSM_tb/dut/key         00101011011111100001010100010110001010001010111011010010101001101010...
/FSM_tb/dut/nextst...   1100
/FSM_tb/dut/oldw        10101100011101110110011011110011 00011001111110101101110000100001 001...
/FSM_tb/dut/rcon        00000001 00000010 00000100 00001000 00010000 00100000 01000000 10000000 ...
/FSM_tb/dut/reset       St0
/FSM_tb/dut/rottemp     01100011000011001010011010110110
/FSM_tb/dut/round       1010
/FSM_tb/dut/round...    xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx...
/FSM_tb/dut/S0          0000
/FSM_tb/dut/S1          0001
/FSM_tb/dut/S2          0010
/FSM_tb/dut/S3          0011
/FSM_tb/dut/S4          0100
/FSM_tb/dut/S5          0101
/FSM_tb/dut/S6          0110
/FSM_tb/dut/S7          0111
/FSM_tb/dut/S8          1000
/FSM_tb/dut/S9          1001
/FSM_tb/dut/S10         1010
/FSM_tb/dut/S11         1011
/FSM_tb/dut/S12         1100
/FSM_tb/dut/state       1100
/FSM_tb/dut/subtemp     01001110111110111111111000100100
/FSM_tb/dut/w           11010000000101001111100110101000 11001001111011100010010110001001 111...
  [0]                   11010000000101001111100110101000
  [1]                   11001001111011100010010110001001
  [2]                   11100001001111110000110011001000
  [3]                   10110110011000110000110010100110
```

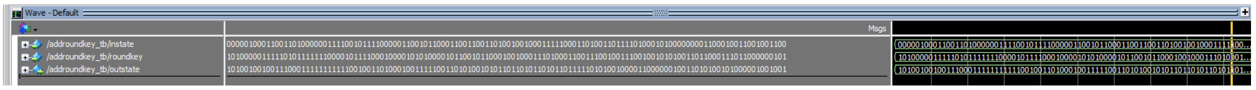
Encryption FSM Testbench Output:

SystemVerilog Code for Add Round Key:



```
// module to perform add round key operation
module addroundkey (
    //input logic [31:0] w0,
    //input logic [31:0] w1,
    //input logic [31:0] w2,
    //input logic [31:0] w3,
    input logic [127:0] instate,
    input logic [127:0] roundkey,
    output logic [127:0] outstate
);

    //outstate[127:96] = instate[127:96] ^ w0;
    //outstate[95:64] = instate[95:64] ^ w1;
    //outstate[63:32] = instate[63:32] ^ w2;
    //outstate[31:0] = instate[31:0] ^ w3;
    assign outstate[127:0] = roundkey[127:0] ^ instate[127:0];
endmodule
```

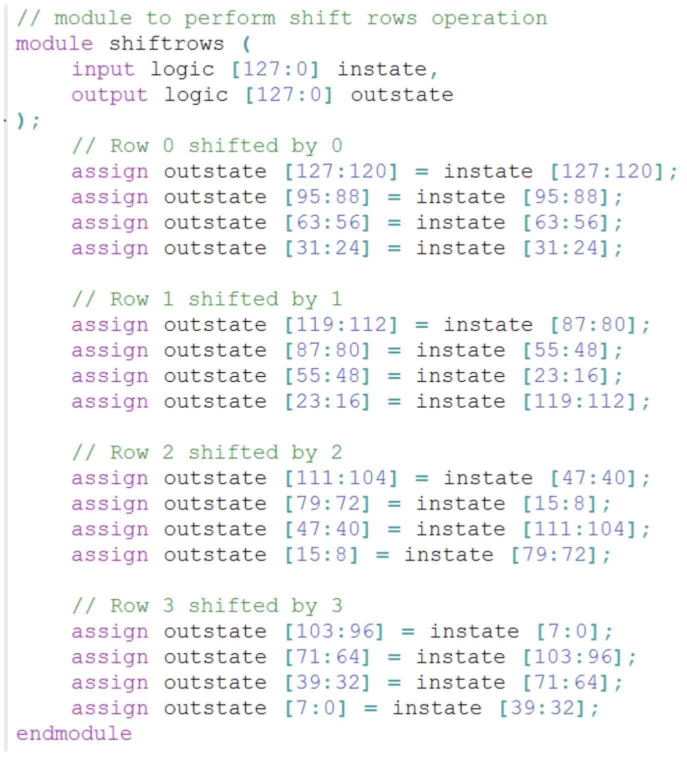
Add Round Key Testbench Output:



```
Wave - Default
                                                                                                                                                Msgs
/addroundkey_tb/instate    00000100011001101000000111100101111000001100101100011001100110100100100011111000110100110111101000101000000001100010011001001100
/addroundkey_tb/roundkey   10100000111110101111111000010111100010000101010000101100101100010010001110100011001110010011100100101010011011000111011000000101
/addroundkey_tb/outstate   10100100100111000111111111100100110100010011111001101010010101101101011010110111110101001000011000000100110101001010000001001001
```

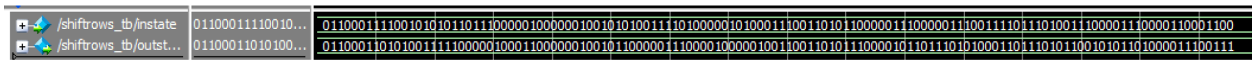
```
00000100011001101000000111100101111000001100101100011001100110100100100011111100...
10100000111110101111111000010111100010000101010000101100101100010010001110100001...
10100100100111000111111111100100110100010011111001101010010101101101011010101101...
```

SystemVerilog Code for Shift Rows:



```
// module to perform shift rows operation
module shiftrows (
    input logic [127:0] instate,
    output logic [127:0] outstate
);
    // Row 0 shifted by 0
    assign outstate [127:120] = instate [127:120];
    assign outstate [95:88] = instate [95:88];
    assign outstate [63:56] = instate [63:56];
    assign outstate [31:24] = instate [31:24];

    // Row 1 shifted by 1
    assign outstate [119:112] = instate [87:80];
    assign outstate [87:80] = instate [55:48];
    assign outstate [55:48] = instate [23:16];
    assign outstate [23:16] = instate [119:112];

    // Row 2 shifted by 2
    assign outstate [111:104] = instate [47:40];
    assign outstate [79:72] = instate [15:8];
    assign outstate [47:40] = instate [111:104];
    assign outstate [15:8] = instate [79:72];

    // Row 3 shifted by 3
    assign outstate [103:96] = instate [7:0];
    assign outstate [71:64] = instate [103:96];
    assign outstate [39:32] = instate [71:64];
    assign outstate [7:0] = instate [39:32];
endmodule
```

Shift Rows Testbench Output:

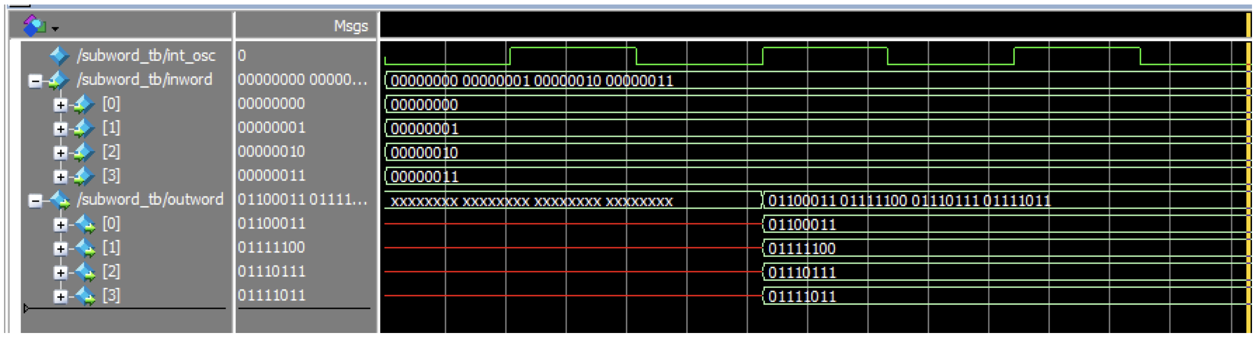


```
/shiftrows_tb/instate   01100011110010...   01100011110010101011011100000100000010010101001111010000010100011100110101100000111000001110011110111010011100001110000110001100
/shiftrows_tb/outst...  01100011010100...   01100011010100111110000010001100000010010110000011100001000001001100110101110000101101110101000110111010110010101101000011100111
```

SystemVerilog Code for Sub Word:

```
// module to perform sub word operation (synchronous using block ram)
module subword (
    input logic [7:0] inword [0:3],
    input logic clk,
    output logic [7:0] outword [0:3]
);
    logic [7:0] sbox_outputs [0:3];

    sbox_sync s0(inword[0], clk, sbox_outputs[0]);
    sbox_sync s1(inword[1], clk, sbox_outputs[1]);
    sbox_sync s2(inword[2], clk, sbox_outputs[2]);
    sbox_sync s3(inword[3], clk, sbox_outputs[3]);

    always_ff @(posedge clk) begin
        outword = {sbox_outputs[0], sbox_outputs[1], sbox_outputs[2], sbox_outputs[3]};
    end
endmodule

module subword_tb (input logic [7:0] inword [0:3], output logic [7:0] outword [0:3]);
    logic int_osc;
    HSOSC #(.CLKHF_DIV(2'b01))
        hf_osc (.CLKHFPU(1'b1), .CLKHFEN(1'b1), .CLKHF(int_osc));

    subword dut(inword, int_osc, outword);
endmodule
```

Sub Word Testbench Output:

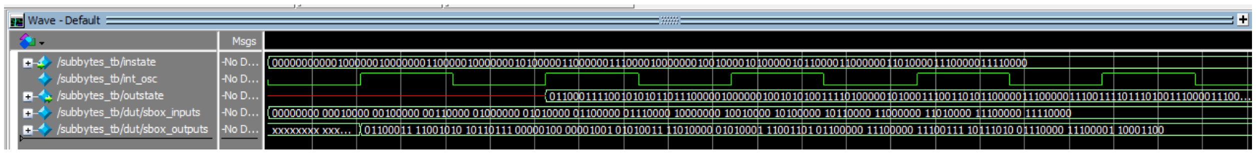


```
                          Msgs
/subword_tb/int_osc       0
/subword_tb/inword        00000000 00000...   00000000 00000001 00000010 00000011
  [0]                     00000000            00000000
  [1]                     00000001            00000001
  [2]                     00000010            00000010
  [3]                     00000011            00000011
/subword_tb/outword       01100011 01111...   xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx   01100011 01111100 01110111 01111011
  [0]                     01100011                                                  01100011
  [1]                     01111100                                                  01111100
  [2]                     01110111                                                  01110111
  [3]                     01111011                                                  01111011
```

SystemVerilog Code for Sub Bytes:

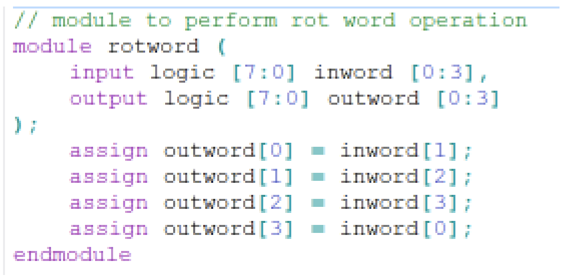
```
// module to perform subbytes (synchronous using block ram)
module subbytes(
    input logic [127:0] instate,
    input logic clk,
    output logic [127:0] outstate
);

    logic [7:0] sbox_inputs [15:0];
    logic [7:0] sbox_outputs [15:0];

    assign sbox_inputs[0] = instate[7:0];
    assign sbox_inputs[1] = instate[15:8];
    assign sbox_inputs[2] = instate[23:16];
    assign sbox_inputs[3] = instate[31:24];
    assign sbox_inputs[4] = instate[39:32];
    assign sbox_inputs[5] = instate[47:40];
    assign sbox_inputs[6] = instate[55:48];
    assign sbox_inputs[7] = instate[63:56];
    assign sbox_inputs[8] = instate[71:64];
    assign sbox_inputs[9] = instate[79:72];
    assign sbox_inputs[10] = instate[87:80];
    assign sbox_inputs[11] = instate[95:88];
    assign sbox_inputs[12] = instate[103:96];
    assign sbox_inputs[13] = instate[111:104];
    assign sbox_inputs[14] = instate[119:112];
    assign sbox_inputs[15] = instate[127:120];

    sbox_sync s0(sbox_inputs[0], clk, sbox_outputs[0]);
    sbox_sync s1(sbox_inputs[1], clk, sbox_outputs[1]);
    sbox_sync s2(sbox_inputs[2], clk, sbox_outputs[2]);
    sbox_sync s3(sbox_inputs[3], clk, sbox_outputs[3]);
    sbox_sync s4(sbox_inputs[4], clk, sbox_outputs[4]);
    sbox_sync s5(sbox_inputs[5], clk, sbox_outputs[5]);
    sbox_sync s6(sbox_inputs[6], clk, sbox_outputs[6]);
    sbox_sync s7(sbox_inputs[7], clk, sbox_outputs[7]);
    sbox_sync s8(sbox_inputs[8], clk, sbox_outputs[8]);
    sbox_sync s9(sbox_inputs[9], clk, sbox_outputs[9]);
    sbox_sync s10(sbox_inputs[10], clk, sbox_outputs[10]);
    sbox_sync s11(sbox_inputs[11], clk, sbox_outputs[11]);
    sbox_sync s12(sbox_inputs[12], clk, sbox_outputs[12]);
    sbox_sync s13(sbox_inputs[13], clk, sbox_outputs[13]);
    sbox_sync s14(sbox_inputs[14], clk, sbox_outputs[14]);
    sbox_sync s15(sbox_inputs[15], clk, sbox_outputs[15]);

    always_ff @(posedge clk) begin
        outstate = {sbox_outputs[15], sbox_outputs[14], sbox_outputs[13], sbox_outputs[12],
                sbox_outputs[11], sbox_outputs[10], sbox_outputs[9], sbox_outputs[8],
                sbox_outputs[7], sbox_outputs[6], sbox_outputs[5], sbox_outputs[4],
                sbox_outputs[3], sbox_outputs[2], sbox_outputs[1], sbox_outputs[0]};
    end
endmodule
```

Sub Bytes Testbench Output:



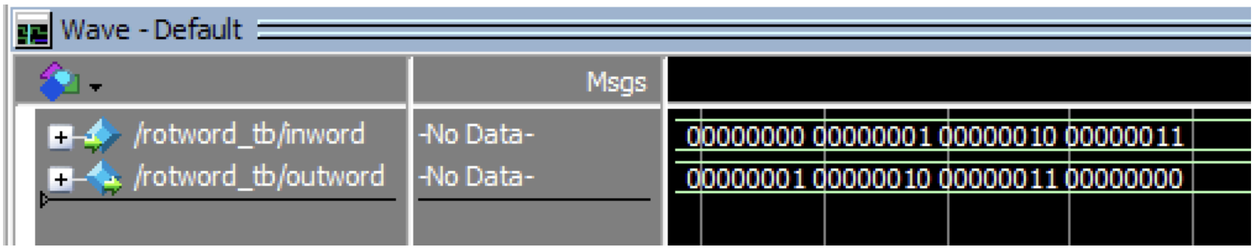
```
/subbytes_tb/instate               -No D...  0000000000001000000100000001100000010000000101000001100000011100001000000010010000101000001011000011000000110100001110000011110000
/subbytes_tb/int_osc               -No D...
/subbytes_tb/outstate              -No D...  011000111110010101011011100000100000010010101001111010000010100011100110101100000111000001110011110111010011100001 1100...
/subbytes_tb/dut/sbox_inputs       -No D...  00000000 00010000 00100000 00110000 01000000 01010000 01100000 01110000 10000000 10010000 10100000 10110000 11000000 11010000 11100000 11110000
/subbytes_tb/dut/sbox_outputs      -No D...  xxxxxxxx xxx... 01100011 11001010 10110111 00000100 00001001 01010011 11010000 01010001 11001101 01100000 11100000 11100111 10111010 01110000 11100001 10001100
```

SystemVerilog Code for Rot Word:



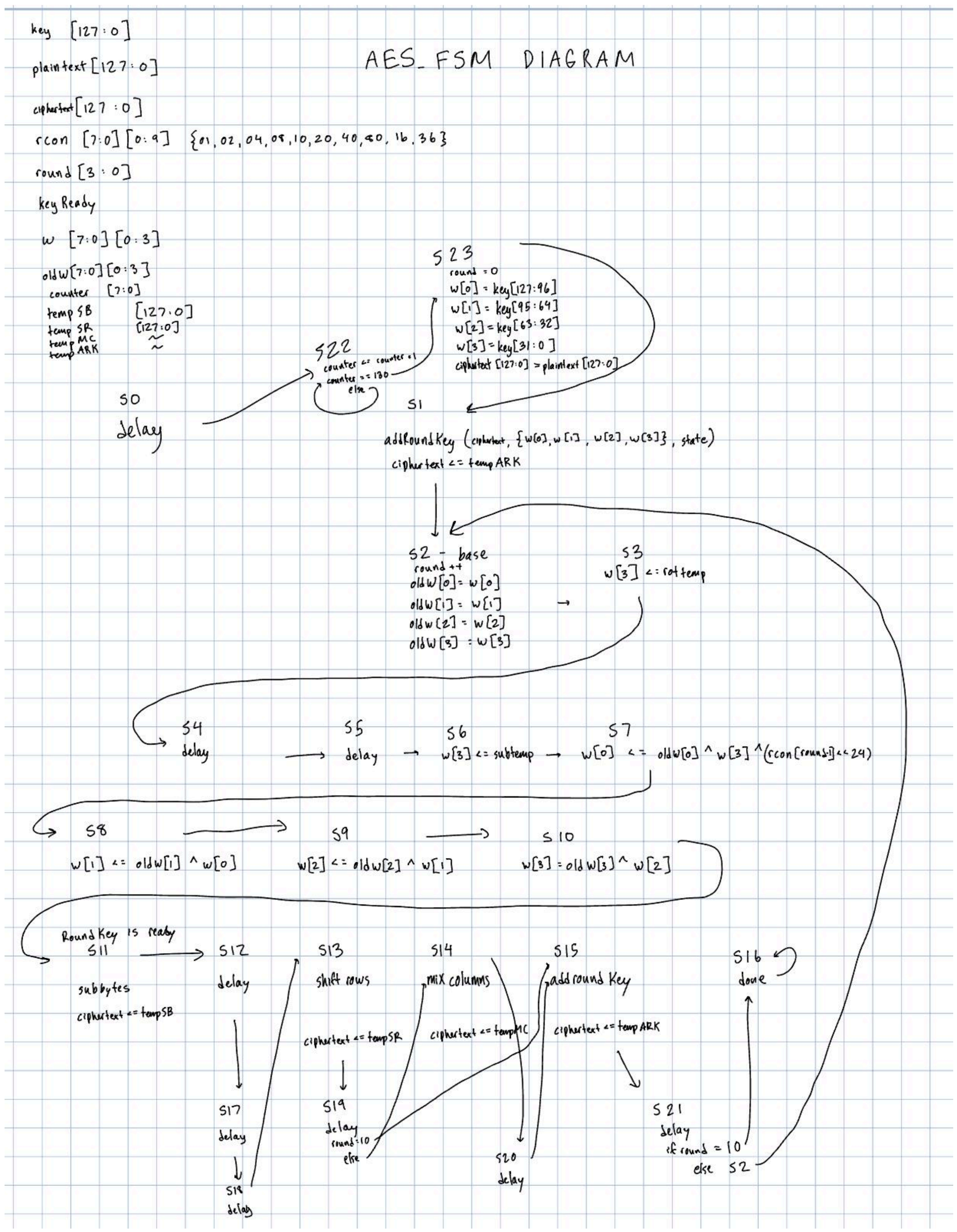
```
// module to perform rot word operation
module rotword (
    input logic [7:0] inword [0:3],
    output logic [7:0] outword [0:3]
);
    assign outword[0] = inword[1];
    assign outword[1] = inword[2];
    assign outword[2] = inword[3];
    assign outword[3] = inword[0];
endmodule
```

Rot Word Testbench Output:

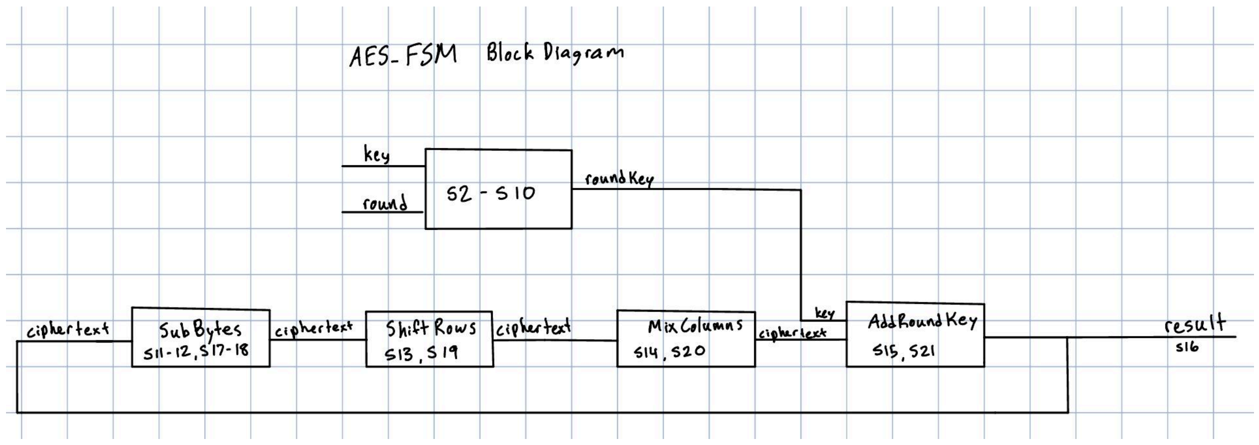


```
Wave - Default
                              Msgs
+ /rotword_tb/inword    -No Data-   00000000 00000001 00000010 00000011
+ /rotword_tb/outword   -No Data-   00000001 00000010 00000011 00000000
```

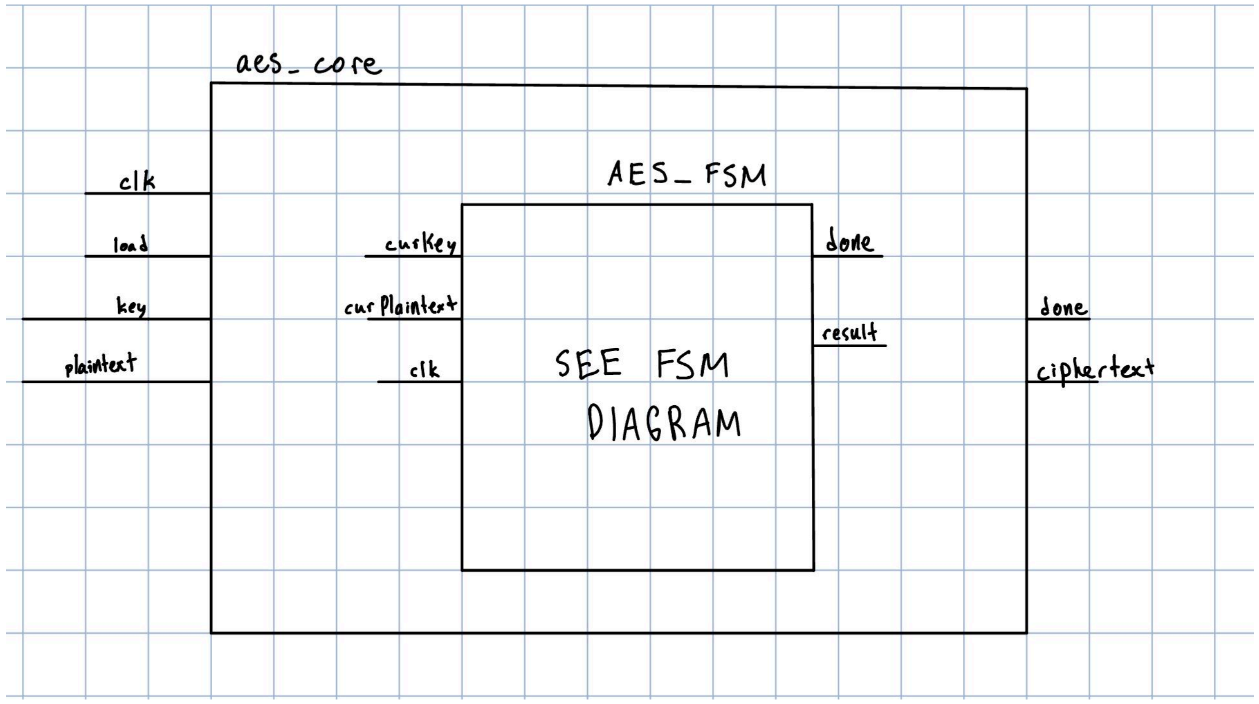
AES_FSM Diagram:



AES_FSM Block Diagram:



aes_core Block Diagram:



Results and Discussion

Throughout this lab, I was able to get very comfortable with each step of AES encryption, including key expansion. The architecture that I designed does fit on the FPGA. It uses 1337 out of the 5280 slice registers and 2443 out of the 5280 4 input Look Up Tables. With this design, I was able to successfully simulate not only the AES encryption given a key and plaintext, but also the core and spi modules that would allow the hardware accelerator to interact with the MCU.

Conclusions

This has been my favorite lab of the class, as I am extremely interested in security and cryptography, especially in hardware. Because of this, I had begun working on implementing AES on an FPGA in the summer, but at the time I barely had any FPGA experience so I struggled with understanding what needed to be done. I had been looking forward to this lab since registering for the course, and it certainly lived up to my expectations. One thing that I found extremely useful for the key expansion part was the AES visualization website, as it was a little hard to understand what was supposed to happen from the pseudocode in the specification (specifically the recursive aspect).

In total, this lab took me approximately 12 hours to complete. At least half of this was spent thinking through and drawing out the FSM to perform key expansion and encryption. The design meets all proficiency requirements.