ECE 385

Spring 2020

Experiment 9

SOC with Advanced Encryption Standard in SystemVerilog

Michael Faitz and Zohair Ahmed

Section ABJ: Friday 2:00-4:50

Yuming Wu and Lian Yu

Introduction

The AES encryptor/decryptor takes a 128 bit message and 128 bit key and gives you both the encrypted and decrypted messages, where the decrypted message is the original message when done correctly. We do this on an Intellectual Property core, which makes reads and writes to a register file in SystemVerilog possible. We handle the process of encryption on the NIOS processor, but decryption in hardware implementation.

Describe the role of the NIOS processor as well as the basic functionality of your C code.

The software encryptor first requires data to be read from the keyboard to obtain the key and plaintext message. We then convert the message and the key into char arrays of hex values. Once that is done, we create a new variable to hold the entire keyschedule, and initialize a message variable, msgChar, to hold the current message. We initialize the values it contains in the function setAllKeys. After we do that, we call roundKey. This function XORs the given msg input with a round key, and stores it in msg. All functions requiring input msg use msgChar. After that we begin a loop subBytes, rowShifter, columnMixer, and addRound key, which executes 9 times. subBytes takes 16 entries, in our case bytes, and converts the 16 entries to their values when converted through the sbox array. rowShifter shifts the rows of the msgchar as needed for the encryption. columnMixer performs the needed column changes on msgchar as needed for the encryption as well. Once the loop ends, we do subBytes, rowShifter, and roundKey. We now have the encrypted message. We store it into the input variable msg_enc, by converting 4 char entries in msgchar to 1 int entry in msg_enc, doing the same for key. This stores the encrypted message and first round key into the register file, allowing it to be used by the decryptor when asked.

Describe the basic steps of decryption and how this is controlled and computed in hardware.

Decryption is handled by the state machine in AES.sv. When instantiating it, we take the key from registers 0 to 3 and the encrypted message from registers 4 to 7. We also take the LSB of registers 14 and 15, the Start and Done registers. These are used to start and pause the state machine for correct variable loading and message storing. The first step we took in decrypting a message is to generate the keyschedule. We set aside 11 clock cycles to allow it to finish before we began the decryption process. After the keys are generated, we did addRoundKey. Then we went through a loop of InvShiftRows, InvSubBytes, AddRoundKey, and InvMixColumns 9 times. We were only allowed one instantiation of InvMixColumns, so we had 4 states for that operation as opposed to one for the other three. After that loop, we did InvShiftRows, InvSubBytes, and AddRoundKey. This process is repeated 10 times with the last time excluding the InvMixColumns. This gives us our decrypted message, which we then store into registers 8 to 11 in our register file.

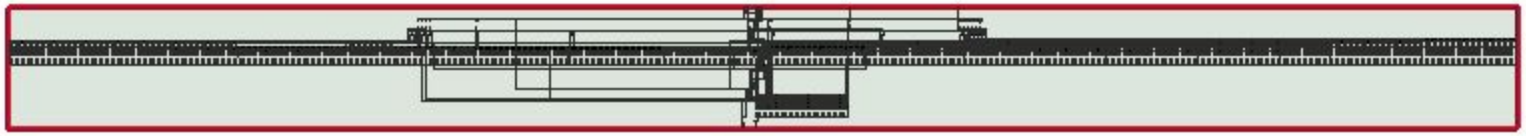
Describe how the system sends data between NIOS and the hardware decryptor and how the register file is designed.

The module takes in the CLK, RESET, AVL_READ, AVL_WRITE, AVL_CS, AVL_BYTE_EN, AVL_ADDR, AVL_WRITEDATA, AVL_READDATA, and outputs a 32 bit register called

EXPORT_DATA. EXPORT_DATA holds the data used for the hex displays. The other signals are used to determine whether or not to write to the register file, as well as what register to write to and which bits to change. The register file is written to using these signals, by using the data in AVL_WRITEDATA which is modified in the NIOS processor. The AVL_WRITE, and AVL_CS signals are also altered in the NIOS processor so that we only write to it when the encryption process is done. AVL_READ is used to check the data used in register AVL_ADDR. The NIOS processor utilizes this by reading in data from the decrypted messages to print it in the terminal. The register file is designed to hold 16 32-bit registers. Registers 0 to 3 hold the key, 4 to 7 hold the encrypted message, and 8 to 11 hold the decrypted message. Registers 14 and 15 hold the start and done registers respectively. The NIOS processor writes to the start register to begin decryption, and checks the done register to confirm that the decrypted message is valid.

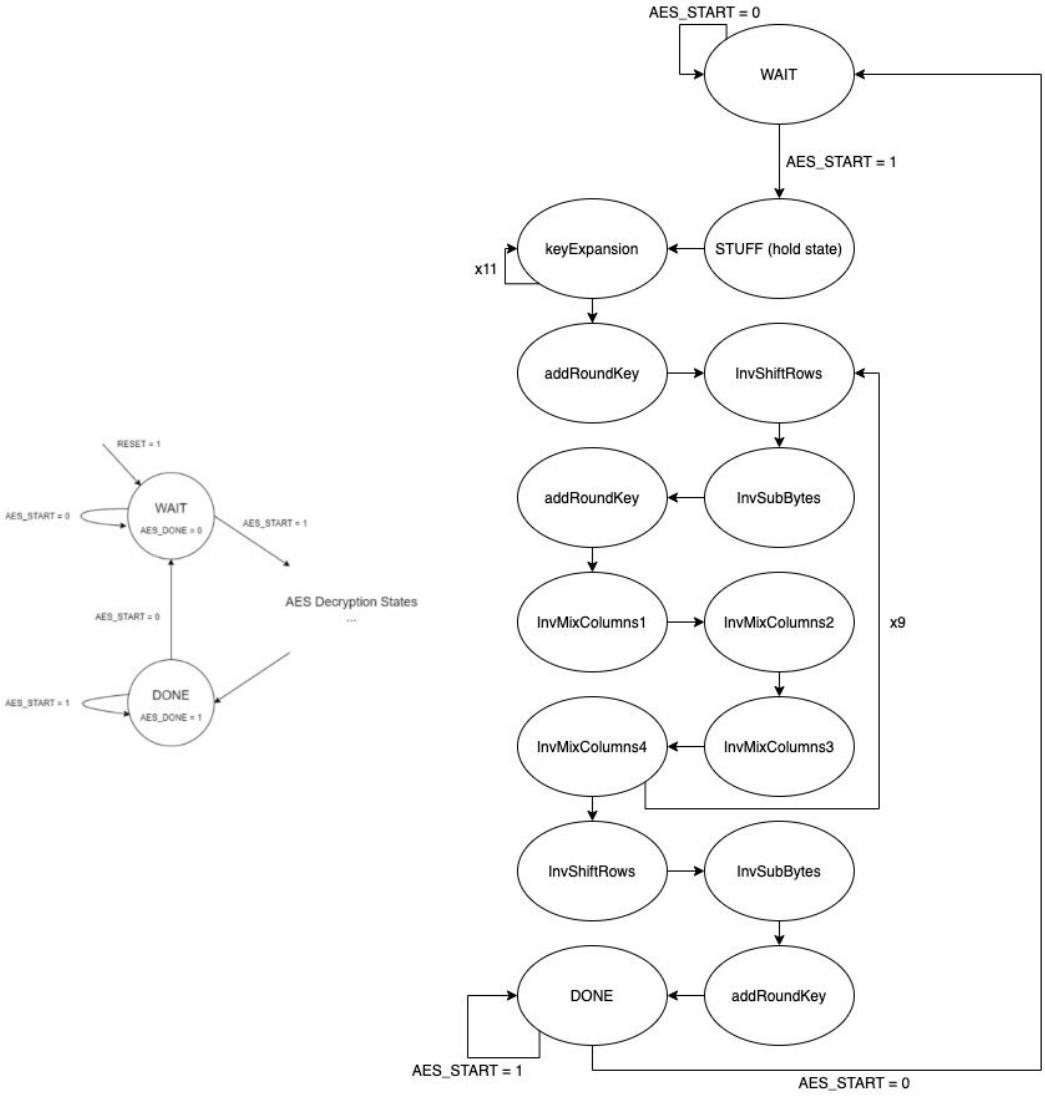
Please include the RTL view of avalon_aes_interface.sv. The Qsys view of the NIOS processor or lab9_top.sv is not necessary for this portion.

This diagram is very large and thus is difficult to get a full screenshot for. Below is the very top level:



State Diagram of AES Decryptor Controller:

- This is the state machine that was written in AES.sv. You may abbreviate the 9 looping rounds in the state diagram like in figure 9 on page IAES.9 of the lab manual



main.c Functions:

Function: void setAllKeys

Inputs: unsigned char * keys, int count

Description: Generates a 128-bit round key upon every function call. keys holds the keyschedule, count is used to denote which key round is being generated.

Purpose: used to generate the keySchedule variable, used for operations in encrypt. Called within a for loop that contains the start and end conditions needed to properly generate keyschedule.

Function: void roundKey

Inputs: unsigned char * round, unsigned char * kptr, int count
Description: Performs XOR operation on round (msgchar) and current round key. As the kptr represents our keySchedule variable, we have a count variable to ensure that we are using the correct round key.
Purpose: Used to perform addRoundKey step in encryption.

Function: void rowShifter
Inputs: unsigned char * m
Description: Shifts rows in m (msgchar) by no shifts in first row, one shift left in row 1, 2 shifts left in row 2, and 3 shifts left in row 3;
Purpose: Used to perform shiftRows step in encryption.

Function: void subBytes
Inputs: unsigned char * currmsg
Description: takes values from a cell, use that value as an index in the aes_sbox, and replace the value in the cell with the aes_sbox at that index. Does this for 16 cells, or one message state.
Purpose: Used to perform subBytes step in encryption.

Function: void columnMixer
Inputs: unsigned char * currmsg
Description: performs matrix multiplication in the form of currmsg * gf_mul and stores the result in currmsg. Performs on 16 cells.
Purpose: Used to perform mixColumns step in encryption.

Function: void encrypt
Inputs: unsigned char * msg_ascii, unsigned char * key_ascii, unsigned int * msg_enc, unsigned int * key
Description: Converts data in msg_ascii and key_ascii from char to int so that it can be used for encryption. Completes full encryption process using above functions, and stores the encrypted message in msg_enc and the first round key in key.
Purpose: Function that sets up variables for encryption, calls each step of encryption, and stores final values into int arrays that are used to write to register file.

Function: void decrypt
Inputs: unsigned int * msg_enc, unsigned int * msg_dec, unsigned int * key
Description: This function writes the key and encrypted message arrays to the register file to prepare it for decryption. The function then sets start register to 0x1, to begin decryption. It waits for the done register to hold 0x1, then reads the decrypted message from register file into msg_dec. This allows it to be printed on the terminal.
Purpose: Function that sets up variables for decryption, begins decryption, and reads decrypted message into array when done for printing on the terminal.

Function: int main
Inputs: none
Description: Asks user to input message and key via keyboard, then prints the encrypted and decrypted message to screen. Also runs benchmark which times the encryption and decryption process and prints the runtimes to the terminal.
Purpose: Runs encryption and decryption process, outputs it to terminal. Also runs benchmark to confirm that encryption and decryption meet expected runtimes.

SystemVerilog Modules:

Module: lab9
Inputs: input logic CLOCK_50, [1:0] KEY
Outputs: output logic [7:0] LEDG, [17:0] LEDR, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5, [6:0] HEX6, [6:0] HEX7, [12:0] DRAM_ADDR, [1:0] DRAM_BA, DRAM_CAS_N, DRAM_CKE, DRAM_CS_N, [31:0] DRAM_DQ, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK
Description: Instantiates lab9_soc from the qsys files and provides the data as well as displays values on the hex displays on the FPGA.
Purpose: Serves as the top level module for this lab.

Module: SubBytes
Inputs: clk, [7:0] in
Outputs: [7:0] out
Description: Large case statement that sinks up each cell within the decrypted message with the table used in AES.
Purpose: Would be used in encryption on each of the temp variables which house the decryption for each cycle of the AES algorithm if that was implemented within hardware..

Module: InvSubBytes
Inputs: clk, [7:0] in
Outputs: [7:0] out
Description: Large case statement that effectively does the same thing as SubBytes, however, the case statements are reversed.
Purpose: Used in decryption on each of the temp variables which house the decryption for each cycle of the AES algorithm.

Module: KeyExpansion
Inputs: clk, [127:0] Cipherkey
Outputs: [1407:0] KeySchedule
Description: Instantiates the module KeyExpansionOne 10 times and builds the keyschedule from the outputs of that function.

Purpose: Generates the keyschedule for the system which is used many times in AddRndKey module for the AES algorithm.

Module: KeyExpansionOne
Inputs: clk, [127:0] oldkey, [7:0] Rcon
Outputs: [127:0] newkey
Description: Generates the next key based on the current key via XORing it with Rcon when needed and the previous key.
Purpose: Used to generate an individual key for use in the larger keyschedule which is essential to the AES method.

Module: InvShiftRows
Inputs: [0:127] in
Outputs: [0:127] out
Description: Creates many temp variables and reverses the shifting operation given within the AES algorithm.
Purpose: Essential part of the process included within each of the 10 loops involved in the decryption process.

Module: InvMixColumns
Inputs: [31:0] in
Outputs: [31:0] out
Description: Instantiates several submodules which handle the unwinding of the column mixing performed in the encryption process.
Purpose: To be instantiated and used on the first 9 cycles of the AES decryption process.

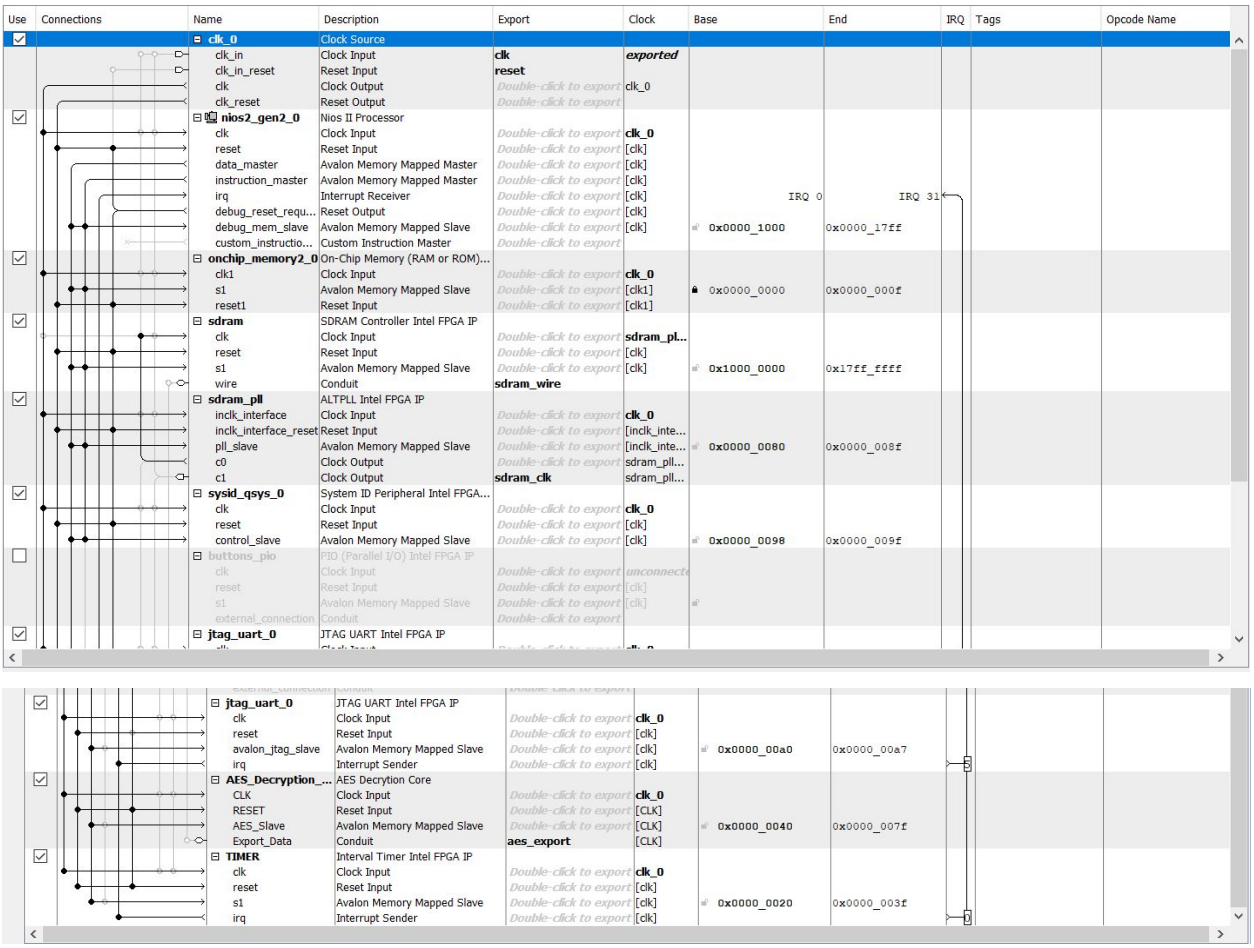
Module: hexdriver
Inputs: [3:0] In
Outputs: [6:0] Out
Description: Maps hex inputs to their seven segment display mode.
Purpose: Used to display the values of the decryption and encryption on the FPGA.

Module: avalon_aes_interface
Inputs: CLK, RESET, AVL_READ, AVL_WRITE, AVL_CS, [3:0] AVL_BYTE_EN, [3:0] AVL_ADDR, [31:0] AVL_WRITEDATA
Outputs: [31:0] AVL_READDATA, [31:0] EXPORT_DATA
Description: Holds the register file and acts as the gate between the hardware and the SystemVerilog program. Holds logic to read and write from various registers in the bank and also instantiates AES.sv.
Purpose: Serves as the primary access point for the SystemVerilog as well as organizes the data for each use in hardware, software, and SystemVerilog. This module also serves to interacts almost completely with the AES_Decryption_Core in QSYS.

Module: AES.sv
Inputs: CLK, RESET, AES_START, [127:0] AES_KEY, [127:0] AES_MSG_ENC
Outputs: AES_DONE, [127:0] AES_MSG_DEC
Description: Serves as the central state machine for decryption as well as instantiates KeyExpansion, InvShiftRows, AddRndKey, InvMixColumns, and InvSubBytes in order to completely decrypt the given encrypted message given the first key.
Purpose: Serves as the main module and does most of the work for the decryption operation.

Module: AddRndKey
Inputs: [127:0] A, [127:0] B
Outputs: [127:0] O
Description: XORs A and B together.
Purpose: To generate the next round key to be used in the cycle given the current decryption and the current key from the keyschedule.

QSYS Modules:



nios2_gen2_0 Block: The nios2 block handles the C code conversion over to system verilog which then is able to be executed within the hardware FPGA board.

Onchip_memory2_0: The memory block acts as available hardware memory for variables needed to execute the C code onto the board. This effectively acts as a compliment to the nios2 processor.

Clk_0 (not shown in screenshot): Serves as the functional clock for the entire system and acts as a reference for the other modules.

sdram: This block allows us to get available SDRAM on the FPGA board since on-chip memory is too small for the available program to be stored and updated successfully. SDRAM is useful due to its quick processing time and low update and output delay.

sdram_pll: This block is the method to account for the small delays within the transfer of the data in and out of the SDRAM and acts as a separate clock for the system.

sysid_qsys_0: This block verifies the correct transfer of the software and hardware by looking back and forth between the C code and SystemVerilog to assure the data transfer is done in the correct format.

jtag_uart_0: Allows for terminal access for use in debugging the software.

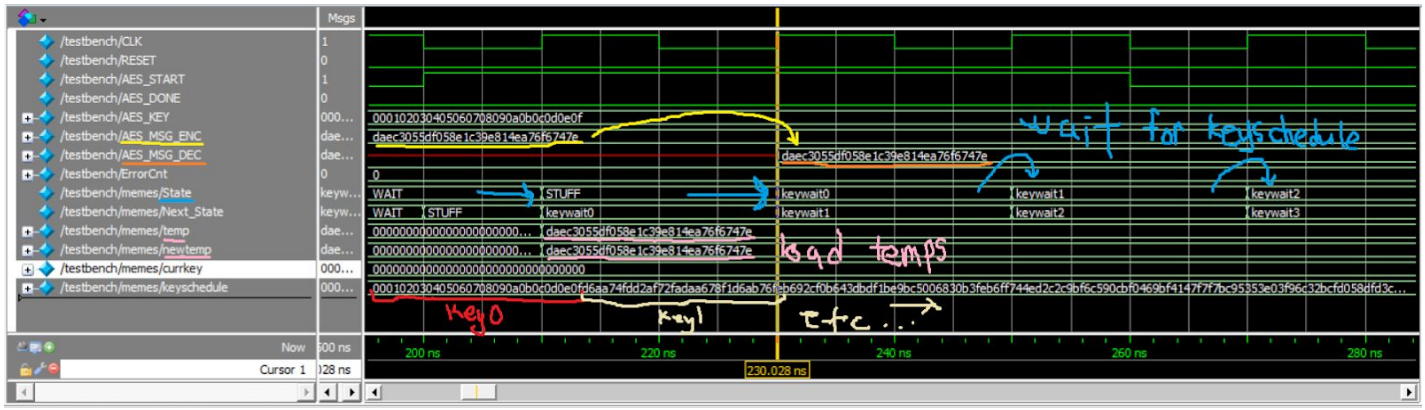
AES_Decryption_Core: This module allows for the input of the keyschedule and the encrypted message onto the hardware which is then processed significantly faster than the software speeds. It also outputs the decrypted message.

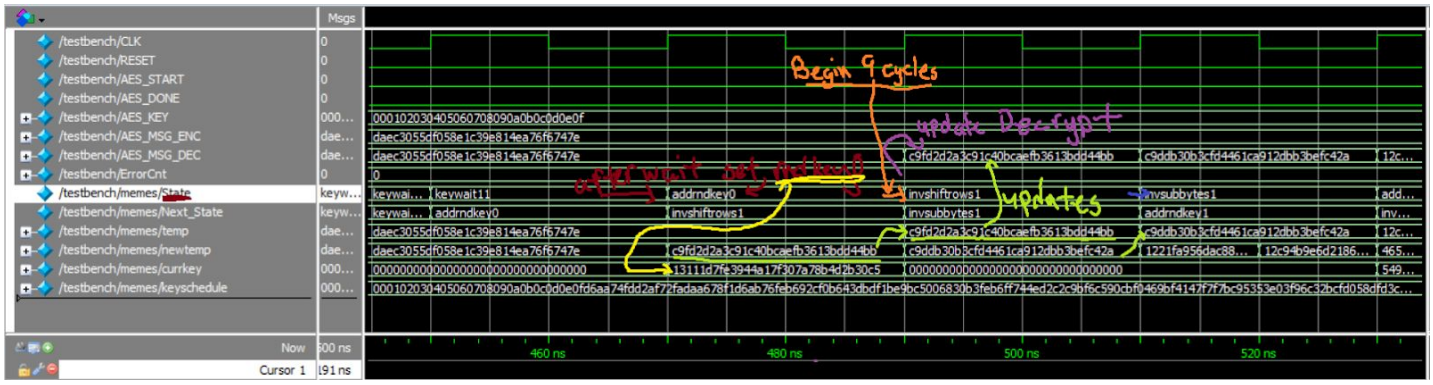
TIMER: Acts as a clock sync for the software to hardware components since software does not have a clock.

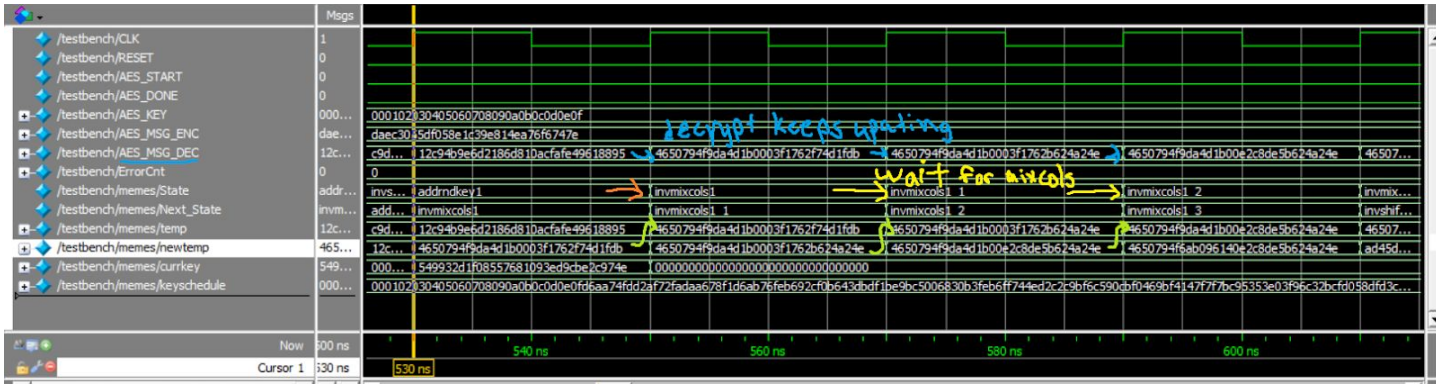
Annotated Simulation of the AES Decryptor

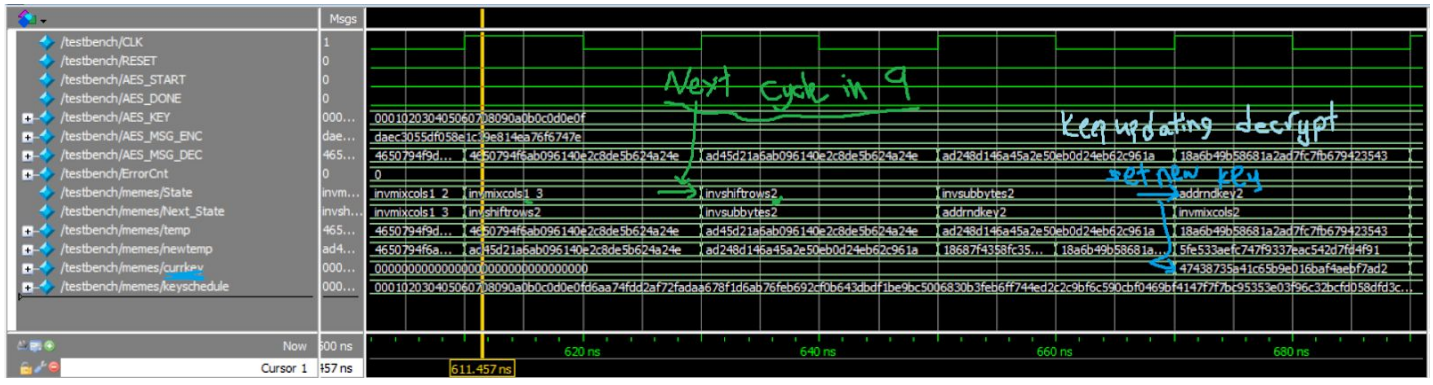
On the diagrams below the input decrypt in plaintext is shown underlined in **yellow** on the first plot and the final decrypted message is shown on the last image circled in **blue**.

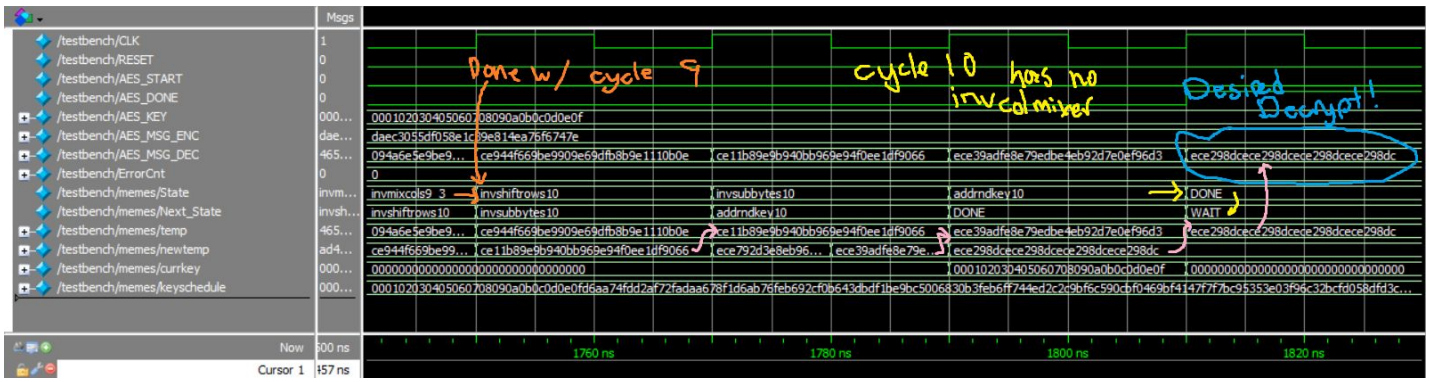
```
module testbench();

timeunit 10ns; // Half clock cycle at 50 MHz
         // This is the amount of time represented by #1
timeprecision 1ns;

// These signals are internal because the processor will be
// instantiated as a submodule in testbench.
logic CLK = 0;
logic RESET;
logic AES_START;
logic AES_DONE;
logic [127:0] AES_KEY;
logic [127:0] AES_MSG_ENC;
logic [127:0] AES_MSG_DEC;

// A counter to count the instances where simulation results
// do no match with expected results
integer ErrorCnt = 0;

// Instantiating the DUT
// Make sure the module and signal names match with those in your design
AES memes(.*);

// Toggle the clock
// #1 means wait for a delay of 1 timeunit
always begin : CLOCK_GENERATION
#1 CLK = ~CLK;
end

initial begin: CLOCK_INITIALIZATION
    CLK = 0;
end


// Testing begins here
// The initial block is not synthesizable
// Everything happens sequentially inside an initial block
// as in a software program
initial begin: TEST_VECTORS

    AES_START = 0;

    AES_MSG_ENC = 128'hdaec3055df058e1c39e814ea76f6747e;
    AES_KEY = 128'h000102030405060708090a0b0c0d0e0f;

    RESET = 1;

    #2 RESET = 0;

    #2 RESET = 0;

    #18 AES_START = 1;

    #6 AES_START = 0;

end

endmodule
```


wait for keyschedule
load temps
key0
key1
etc...
230.028 ns


Begin 9 cycles
update Decrypt
Updates


decrypt keeps updating
wait for mixcols
530 ns





Fill out the design resources and statistics table.

LUT	6014
DSP	0
Memory (BRAM)	126080
Flip-Flop	2948
Frequency	119.56 MHz
Static Power	102.29 mW
Dynamic Power	0.73 mW
Total Power	174.05 mW

Which would you expect to be faster to complete encryption/decryption, the software or hardware? Is this what your results show?

We would expect decryption to be faster because it is all handled on the hardware, whereas encryption happens on the NIOS chip, or software. When the encryption process is done, the NIOS chip needs to write the encrypted message to the register file. This data transfer requires many clock cycles and slows down the process significantly.

```
Select execution mode: 0 for testing, 1 for benchmarking:
1
Software Encryption Speed: 0.608828 KB/s
Hardware Encryption Speed: 222.222222 KB/s
```

If you wanted to speed up the hardware, what would you do?

To speed up the hardware, one thing we could do is handle key expansion while the rest of the state machine operates. While this was not specifically restricted by the lab instructions, we

chose to handle that before any decryption operations. Doing those operations in parallel would speed things up. Another thing we could have done was instantiate InvMixColumns 4 times, which would allow us to do that decryption step in one state instead of 4.

Discuss functionality of your design. If parts of your design didn't work, discuss what could be done to fix it.

Our design succeeded in creating a state machine that handled decryption, wrote it to the register file and displayed the first and last two bytes on the hex displays. We were able to display the correct encrypted and decrypted message in the NIOS terminal and got expected benchmark values. We were not, however, able to run consecutive decryptions correctly. We suspect this to be an error in changing one of the values needed to run the steps correctly. We would need to run the code in simulation and confirm that the initial values are correct, then check keyExpansion to make sure that we are not altering the Rcon table when expanding keys.

Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right so it doesn't get changed.

A note explaining how to convert the char arrays to the plaintext and vice versa would have been helpful, as well as converting the Rcon array to a format we could use would have been helpful. We figured out that we needed to right shift 24 bits in keyExpansion but it took us a fair amount of time to figure out. Other than that, nothing was horribly unclear. The intermediate steps in the given .txt file was incredibly helpful and helped us get our demo points. One for the other test case could be helpful.