ECE 385

Spring 2020

Experiment # 9

SOC with Advanced Encryption Standard in SystemVerilog

Megh Shah, Ishaan Patel
Lab Section: ABC, Tuesday 11:30 AM
TA: Gene Shiue, David Zhang

Introduction:

In this lab, we explored the ideas of encryption and decryption. The purpose of these processes is to allow a secure end-to-end communication between two parties where only the two communicators are able to decipher the text and read the message. The encryption operation works by taking a cipher key and a plaintext message, and by using the cipher key, various forms of scrambling and substitutions are applied to the message which creates a completely new message. For the decryption process, the encrypted message and the cipher key are given, and the same methods of scrambling and subsitions are applied, however, this time in reverse to produce the original plaintext. For our lab, we applied these operations and methods to encrypt and decrypt 128-bit keys which ultimately help us learn more about both software and hardware concepts.

Written Description and Diagrams of the AES encryptor/decryptor:

a. **Written Description of the Software Encryptor** –
Regarding the functionality of the software encryptor, we utilized both the NIOS II processor and our C code. The software encryption works taking two inputs: the plaintext message and the cipher key. These inputs are passed in through the console which store the message and the key in their respective registers in the NIOS II processor. From there, our C code accesses these registers and converts the message and key from ASCII form to HEX form. Using the HEX forms, we then call on the AddRoundKeys function with the initial cipher key, or the first roundkey, with the plaintext message. The AddRoundKeys function takes one of 11 generated round keys, which are pre-computed using previous round keys, and XORs it with the current message. After this, we call the SubBytes function, ShiftRows function, MixColumns function, and the AddRoundKey function again in that order on the current message. The SubBytes function takes a pair of bytes and substitutes it with another pair of bytes from the S-box. The ShiftRow function takes each row of the 4x4 matrix of the current message and shifts them around uniquely to scramble the message. The MixColumns function operates in a similar way, but with columns and also applies linear transformations to further scramble the columns. This process is repeated nine times, and thereafter, we call the SubBytes function, ShiftRows function, and AddRoundkey function one last time. The message that is the result after this process is the encrypted message, and is stored in the encrypted message registers on the NIOS II processor.

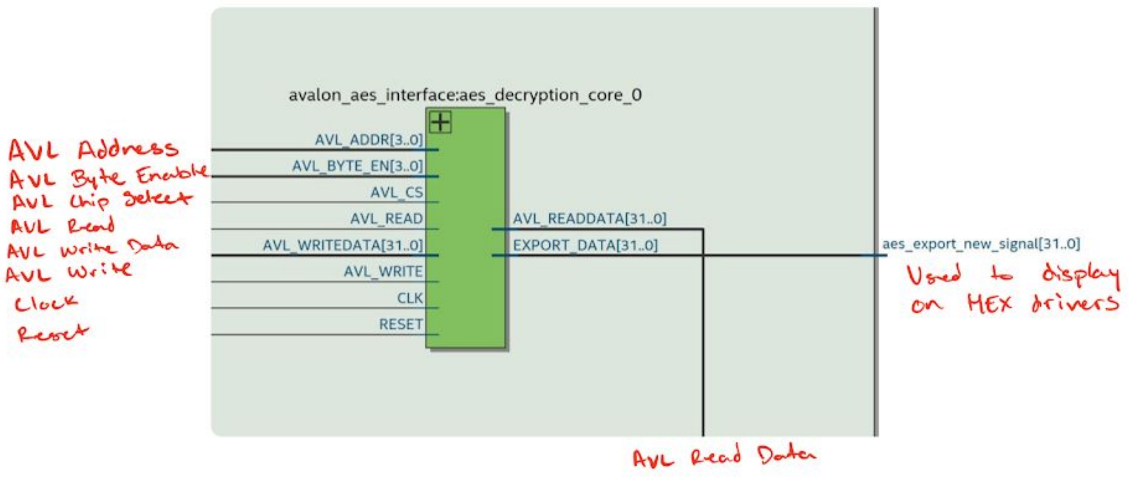
b. **Written Description of the Hardware Decryptor** –
The hardware decryptor works in a similar way as the software encryptor. It follows the same process and executes the same functions, however, this time in reverse. For our decryptor, we created multiple MUXes and extra registers. Essentially, we have a main MUX whose inputs are the outputs of the various

different functions, such as SubBytes, ShiftRows, and AddRoundKey. The control signal for this MUX is directed by the state of the finite state machine. So, for example, if we were in the state that was supposed to substitute the bytes, that state would set the control signal for the MUX to choose the SubBytes input and would store that output in our temporary register which holds the current message. For the InvMixColumns function, since we could only instantiate it once, we worked around that by passing in only 32-bits at a time, or one column, and then storing those 32-bits in a temporary register. We repeated this three other times with the other columns, and then would concatenate all the bits together before sending it to the current message register. All in all, our hardware decryptor works by changing the select line to our main MUX which chooses the correct input depending on the FSM. The FSM contains a tracker which allows it to loop nine times during the middle portion of the process. Once the process is done, the plaintext is stored in the message decrypted registers on the NIOS II processor.

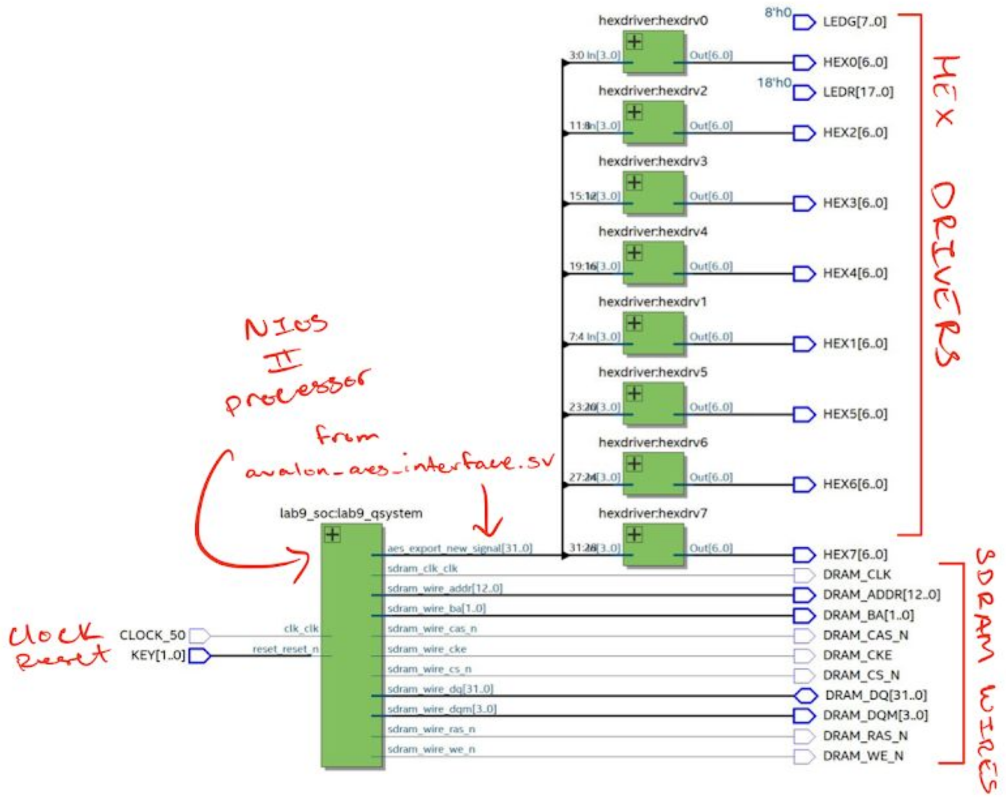
c. **Written Description of the Hardware/Software Interface** –
The system sends data between the NIOS II processor, the software C code, and the hardware components for the decryptor by storing and accessing data in the registers in the register file. The register file is designed to contain 16 different 32-bit registers. Registers zero to three contain the 128-bit cipher key in 32-bit increments. Registers four to seven contain the encrypted message and registers eight to eleven contain the decrypted message in the same fashion. Registers twelve and thirteen are not used. Register fourteen contains the AES_START bit and register fifteen contains the AES_DONE bit. These registers are utilized by both the software and hardware. After the console takes the inputs for the key and message from the user, it stores it in the respective registers in the register file. Both the software and hardware then access these registers to read the key and message, perform their necessary computations and functions, and then store the result in the correct registers, such as the encrypted or decrypted registers. The AES_START and AES_DONE registers are only used to signal whether the process of encryption or decryption should begin or is completed.

d. **Block Diagrams** –

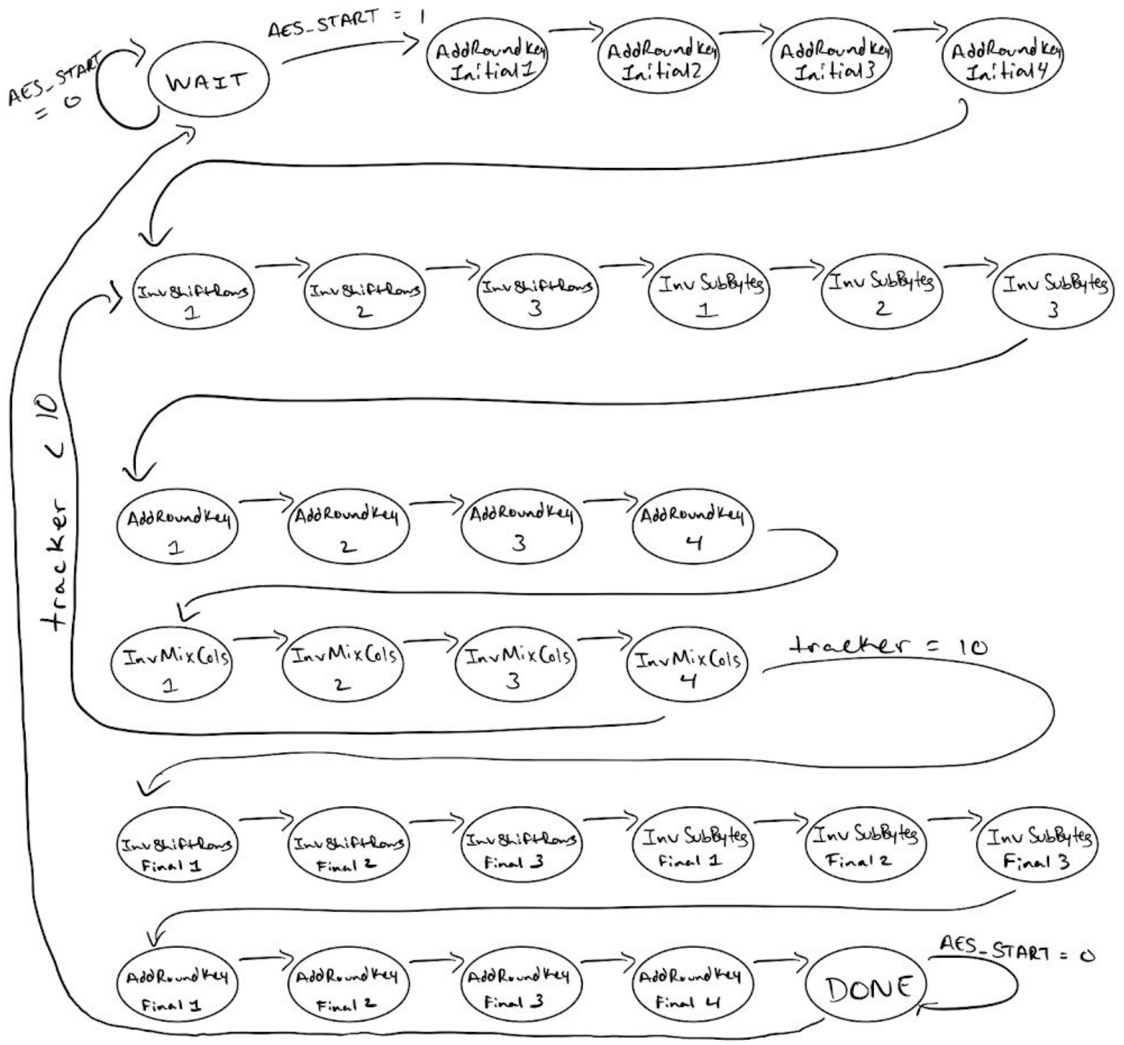
i. RTL View of avalon_aes_interface.sv:



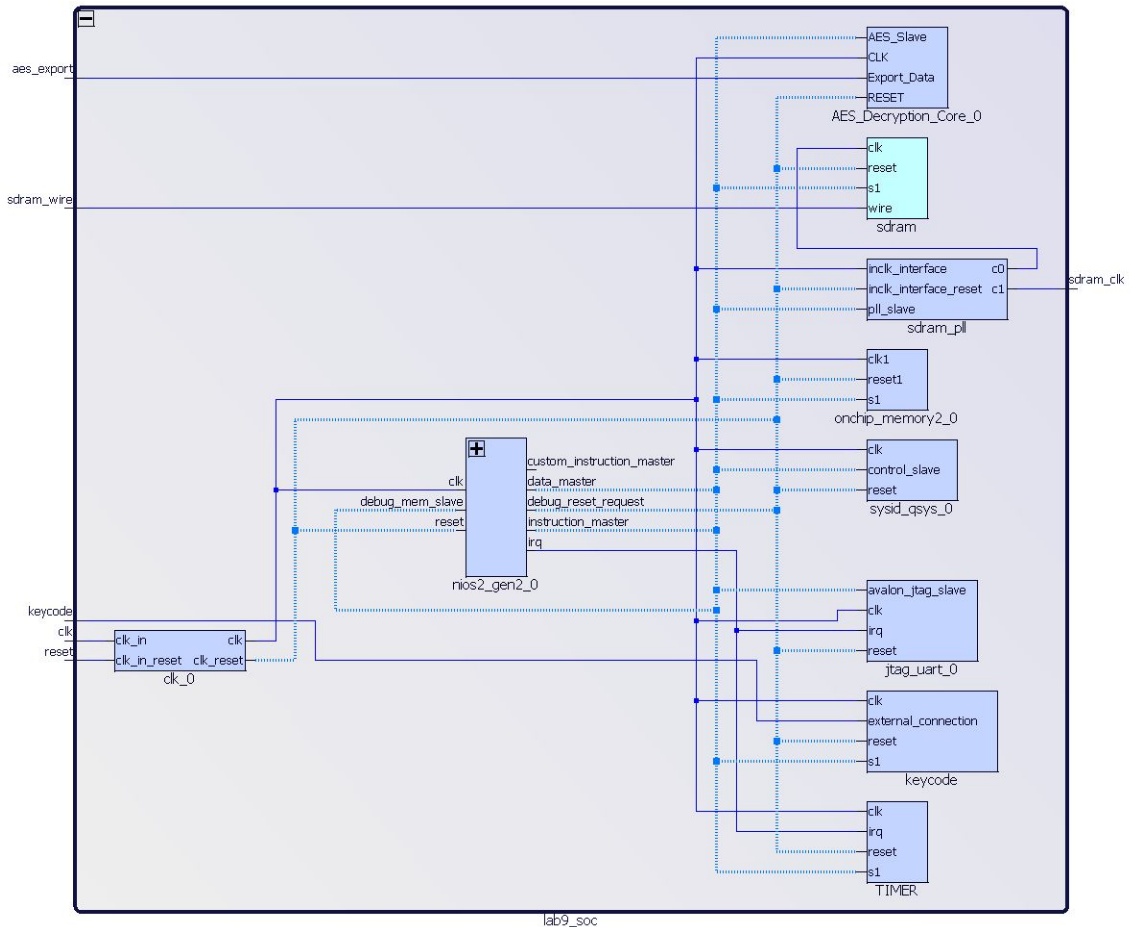
ii. Top-Level RTL Diagram (lab9_top.sv):



e. **State Diagram of AES Decryptor Controller** –



Qsys View of NIOS II Processor –



Module Descriptions:

Hex Drivers - Used to display data on the HEX displays on the FPGA board

Keycode - Contains the ASCII value of the keypress on the keyboard, and is exported so the data is accessible to other hardware

SDRAM Controller – Contains instructions which the NIOS II processor executes from

SDRAM Clock – Special clock used for the SDRAM to avoid metastability and propagation delays

On-Chip Memory – Used to store data and values. Needs to be constantly refreshed or else will lose data

System ID Checker – Checks whether the hardware and software are compatible

Timer – Used in the C program to measure start and end times of the encryption and decryption processes

KeyExpansion – Takes in a 128-bit cipher key and produces 11 roundkeys which is stored in 1408-bit data

InvSubBytes – Takes in a pair of bytes and returns it substitution from the inverse S-box

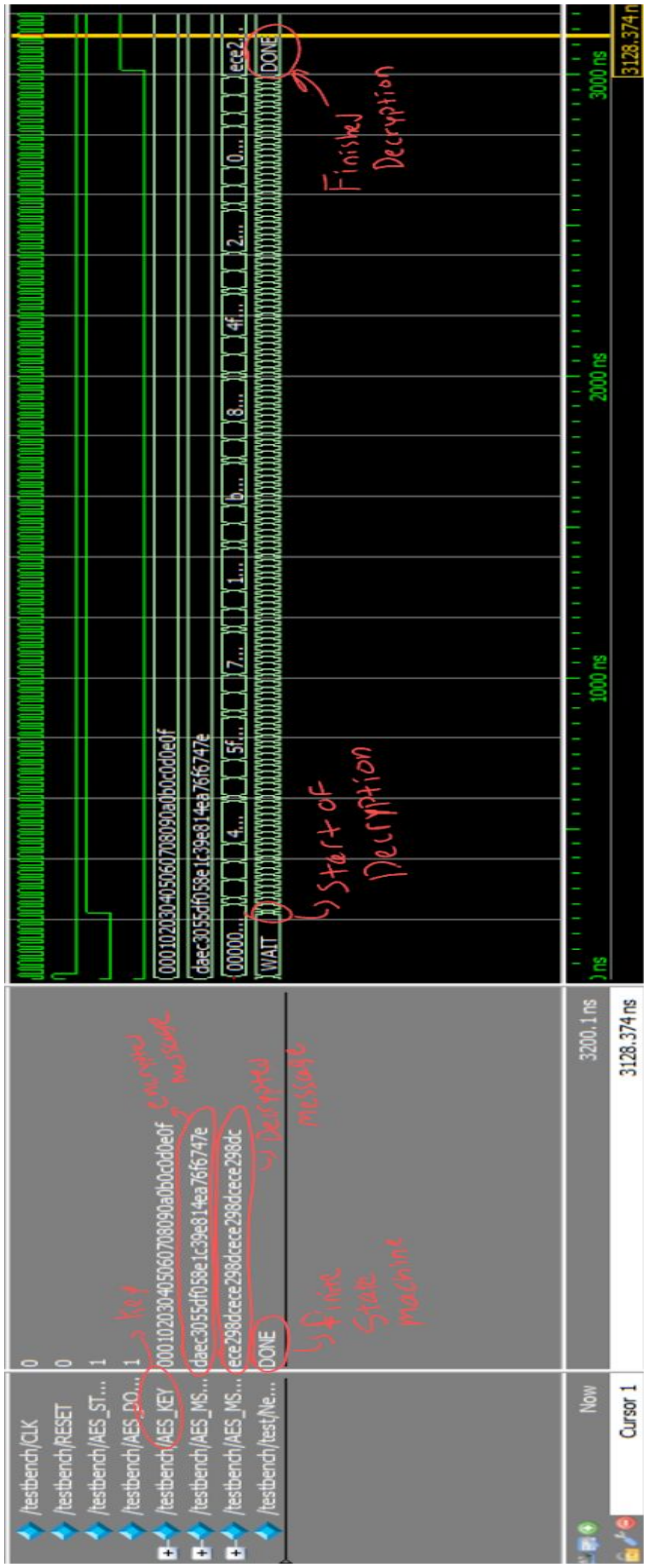
InvShiftRows – Takes in 128-bit data and shifts each row uniquely in column-major order

InvMixColumns – Takes one column, 32-bit input, and applies linear transformation as well as shifting the bytes in each column

avalon_aes_interface – The interface for the avalon decryption core in the NIOS II processor that allows the storing and accessing of data in the registers

AES – Contains all MUXes, temporary registers, and finite state machine in order for the decryption process to operate as intended

Annotated Simulation of AES Decryption



Post-Lab Questions:

LUT	7,054
DSP	0
Memory (BRAM)	571,392
Flip-Flop	4,327
Frequency	65.82 Mhz
Static Power	102.64 mW
Dynamic Power	77.66 mW
Total Power	258.70 mW

1. Which would you expect to be faster to complete encryption/decryption, the software or hardware? Is this what your results show? (List your encryption and decryption benchmark here)
    a. Hardware is faster than software.
    b. Encryption: 0.424268 KB/s
    c. Decryption: 222.2 KB/s

2. If you wanted to speed up the hardware, what would you do? (Note: restrictions of this lab do not apply to answer this question)
    a. One way to speed up hardware is by running the modules in parallel. This would increase our throughput significantly. We can see this in action with inverse subbytes, where we instantiated 16 modules in parallel.

Conclusion:

Regarding the functionality of our design, our design worked mostly as intended as we prepared beforehand. We tested each function of encrypt and decrypt individually before putting the entire process together to eliminate errors in the beginning. One issue we did run into was our MUX that is responsible for choosing the correct round key from the key schedule was wired up wrong. Due to this, the MUX always defaulted to the very first input and caused the original cipher key to be used neglecting which loop it was in. However, this was a simple fix as we just had to rewire the MUX and then it worked as intended. Regarding the lab manual and the instructions for this lab, we believe that this lab was very straightforward and there was little to no ambiguity. This lab had many detailed guides and instructions which we were able to follow to complete the lab. Overall, by completing this lab, we believed that not only did we learn an interesting concept regarding computer security, but we learnt more regarding software and hardware intercommunication and how one has its benefits over the other.