# ECE 385
# Spring 2019
# Experiment #9

# SOC With Advanced Encryption Standard in SystemVerilog

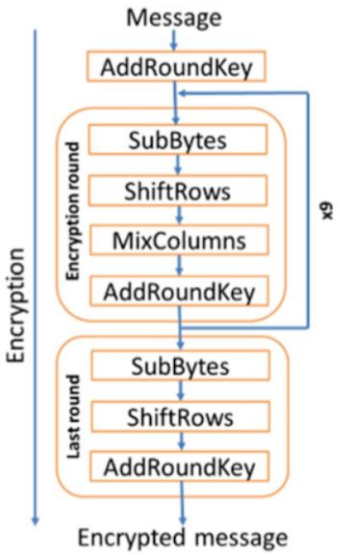
Arkin Shah (arkinas2)
Brian Zheng (brianjz2)
Thursday 8am (ABL)
TA: Vikram Anjur
Xinying Wang

**Introduction**

This was the final experiment of the semester and in this we implemented the Advanced Encryption Standard on the FPGA using SoC technology. The AES Encryption standard will take a message and a key, process the key to create a cipher to encrypt that message. Likewise, the key can be used to also decrypt the encrypted message back its original form. This lab saw us dive into understanding how the AES algorithms work and the way it can be implemented in both the NIOS II Processor (software) for encryption, and the FPGA hardware for decryption.

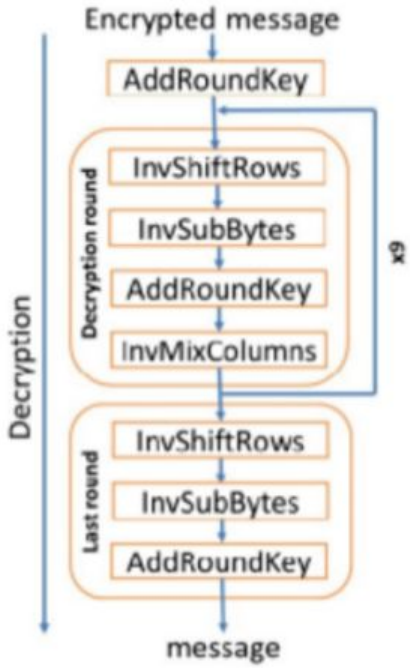
**Software Encryption:**

The NIOS II processor is responsible for processing the message and encrypting. The various c functions, are run by the NIOS II to be displayed on the HexDrivers. The encryption is done by a function that performs the AES Encryption algorithm as seen below. Not pictured on the algorithm is the Key Expansion, which is needed to expand the key that is implemented by the user for encryption. Including the key expansion function,  the C encryption function calls five functions (note that key expansion is called before the first AddRoundKey at top). The way we implemented our software encryption involved placing the two 8-bit char arrays (key and message in ascii), using a function to convert those to hex and placing them into two 16 bit arrays, performing the encryption functions, and then subsequently shifting back the values to get our encrypted message. We then have our message displayed on the HexDriver by having the first four bits of the AES pointer contain the value of the key and the next four carry the encrypted message. This is how the message is also sent to the hardware for decrypting. The overall functionality of our software encryption is fully operational and yields the correct value, though we weren't able to get it to function until after demonstrations were over.



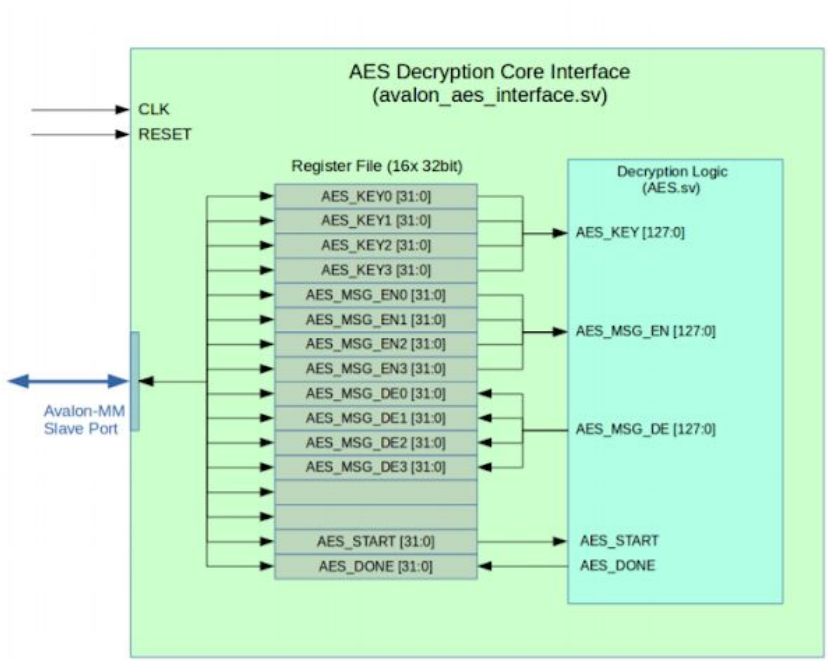
**Hardware Encryption:**

The AES_PTR which was outlined in the previous section, the hardware in this lab performs the decryption. Unlike the encryption, the decryption is all done on hardware, with the algorithm below performed via modules on the FPGA. The various modules that were given to us included the KeyExpansion, AddRoundKey, InvShiftRows, and InvSubBytes. For our implementation, the circuit will perform an addroundkey, a invShiftRows, and invMixColumns, and an invSubBytes module simultaneously. Via a state machine and 4 to 1 mux that we implemented into the hardware, we will select the correct signal to implement into the register module (which holds the value of currentValue, our desired message). This is performed nine times as shown on the diagram (and subsequently currentValue is also changed as well per cycle) before performing the last round and finally placing currentValue onto AES_DECRYPT when AES_DONE is set to 1.

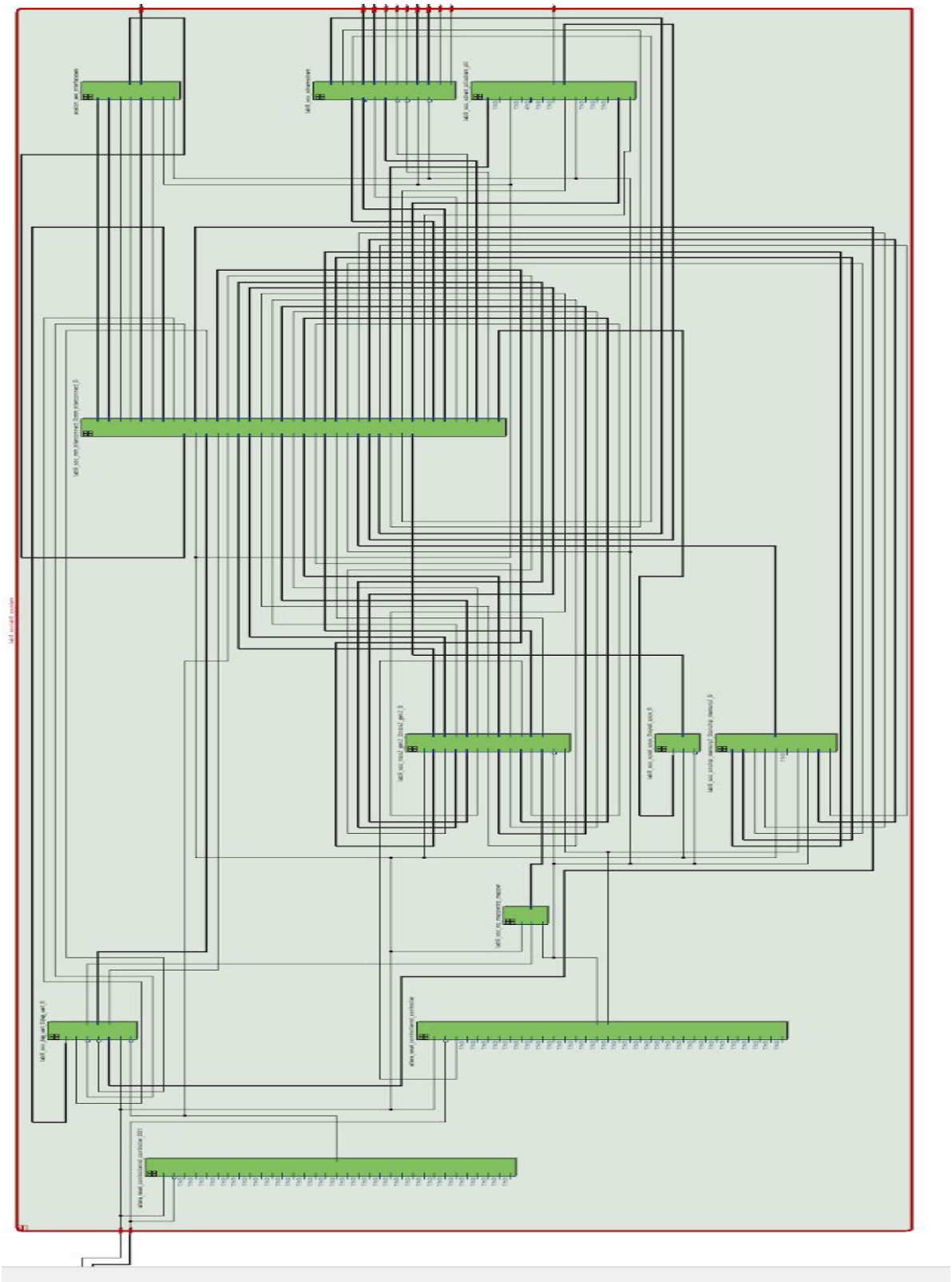


**AES_AVALON_INTERFACE:**

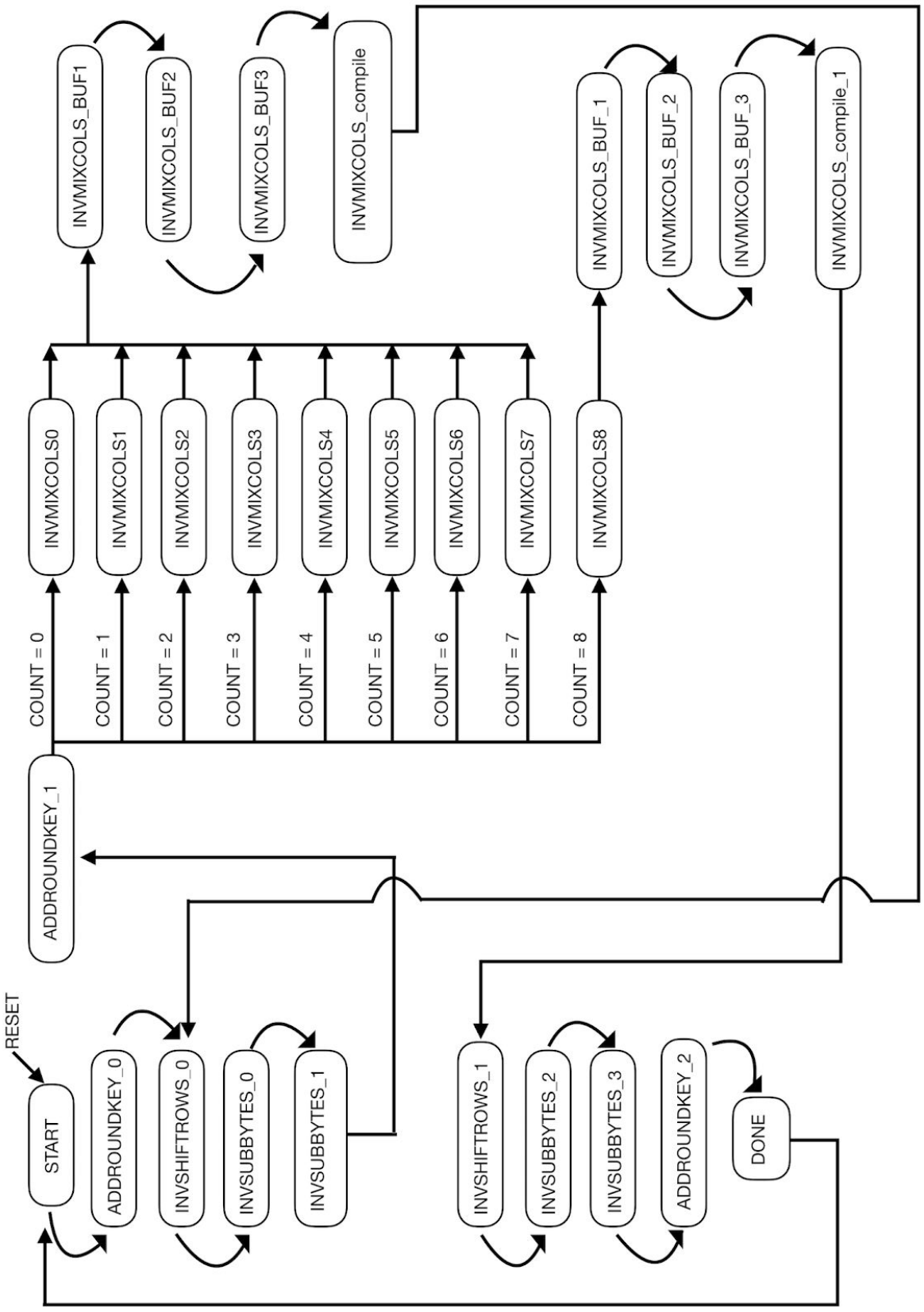
This program is important as it serves as the communication point between the NIOS II Processor and the FPGA hardware. In particular, as seen in the diagram below, the avalon consists of 16x32bit registers that can hold the key, encrypted message, decrypted message, and control signals to process whether to continue or stop encrypting (via AES_START and AES_DONE). When the software needs to fetch the decrypted message, it will make a call to AES_PTR [8-11] to fetch the message. Also shown in the diagram, the avalon is also responsible for instantizing the AES.sv file, which contains the modules needed to perform the decryption and send the message. Once AES.sv is finished performing the function, AES.sv will place the decrypted message into AES_PTR[8-11] while subsequently changing the AES_START and AES_DONE signals to signal to the software that decryption is complete.



## Block Diagram



**Decryption State Machine**



**Module Descriptions:**

Module: lab9_top
Inputs: CLOCK_50, [1:0] KEY,
Outputs: [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, RAM_CS_N, [3:0] DRAM_DQM, DRAM_RAS_N,DRAM_WE_N, DRAM_CLK
Inout: [31:0] DRAM_DQ,
Description: Top-level function for the experiment
Purpose: Instantizes the entire experiment, including the processes needed for the avalon to process and decrypt a message, and the NIOS system for software encryption.

Module: aes_avalon_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: Interface to communicate between hardware decryption and software encryption
Purpose: Via the usage of a pointer specified at a specific address, the avalon will take the encrypted message and key, process it using the algorithm and then send it back to the software for display. (More on that in the avalon_interface section)

Module: AES
Inputs: CLK, RESET, AES_START, AES_DONE, [127:0] AES_KEY, [127:0] AES_MSG_ENC,
Outputs: [127:0] AES_MSG_DEC
Description: Instantizes the AES Decryption module
Purpose: Called by the aes_avalon_interface to perform the decryption. All modules needed for decryption are called in this module.

Module: register #(parameter n = 128)
Inputs: logic clk, [n-1:0] din,
Output: [n-1:0] dout
Description: Basic Register to hold a value
Purpose: For the way we implemented the decryption, we needed to have the the incomplete decryption message continue to cycle through. The register module is used to prevent latching errors.

Module: addroundkey
Inputs: [127:0] key, [127:0] in,
Outputs: [127:0] out

Description: Performs an XOR function given a key and in. This is fed to output
Purpose: Implements the XOR function for the AddRoundKey.

Module: fourtoonemux
Inputs: (default parameter n = 128)
input logic [n-1:0] in, in1, in2, in3,
input logic [1:0] select,
Outputs: [n-1:0] out
Description: Takes in 4 inputs, a 2 bit select signal and throws out a singular output selected from the inputs
Purpose: Used when compiling results from inverse mixcolumns and also used to select which signal to pick considering what state we are in (i.e addroundkey, invsubbytes, invshiftrows or invmixcolumns) since all 4 modules are always calculating and processing the signals, however which result to use is picked by this MUX which gets its select bits from the finite state machine module. It is also called to break down and send the 128-bit signal to the 32-bit processing module for inverse mix columns.

Module: tentoonemux
Inputs: [127:0] in1, in2, in3, in4, in5, in6, in7, in8, in9, in10, [3:0] select
Outputs: [127:0] out
Description: Takes 10 input signals and a 4 bit select signal to throw out a singular output
Purpose: Used to choose which signal goes into the main bus from the ADDROUNDKEY moudule as there are 10 different keys produced and a different key is used for each round.

Module: twotoone
Inputs: [127:0] in, in1,  select
Outputs: [127:0] out
Description: Basic two-to-one mux
Purpose: Used in the beginning of the decryption operation, whether to select the beginning value AES_ENCRYPT or currentValue, which is AES_ENCRYPT processed by the AES decryption algorithm at a certain point in time (selects AES_ENCRYPT only during first run since we can’t change AES_ENCRYPT).

Module: weirdRegister
Inputs: clk, [1:0] select,  [31:0] din
Outputs: [127:0] out
Description: Takes the result of InvMixColumns and selects the correct set within for the cycle.
Purpose: Based on the way we implemented the Decryption circuit in verilog, the decryption needs to be able to grab the correct column to be used. This is where this weird register comes in. Based on the 2-bit select it will grab the specific part of the 128 bit output out is for the respective cycle and assign din to that part. It acts both like a MUX and a register at the same time.

Module: FSM
Inputs: CLK, RST, getAES_START, getAES_DONE,
Outputs: [3:0] round, [1:0] REGMUX, [1:0] MIXCOLMUX, valMux
Description: State machine for the AES decryption
Purpose: This is the brains of the AES decryption cycle. It takes into consideration the various muxes needed to process the operations of the system. It helps the system move from one state to another by sending the valid output control signals. Once it gets the start signal, it is pretty much autonomous till it ends.

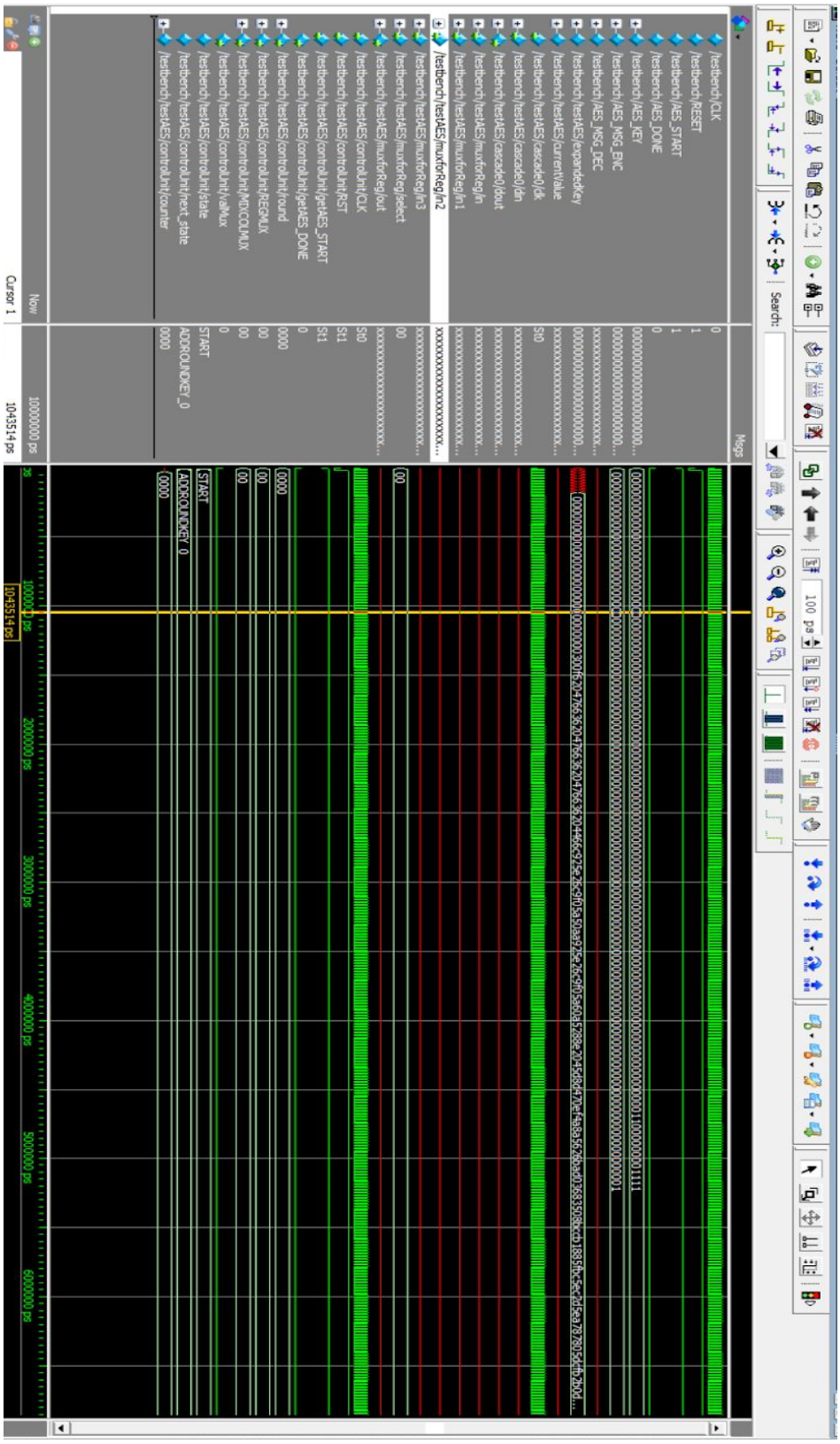
Module: Key Expansion
Inputs: clk, [127:0]  Cipherkey,
Outputs: [1407:0] KeySchedule
Description: Verilog implementation of the KeyExpansion function
Purpose: Takes a 128-bit key, and implements the KeyExpansion onto a 1408 long KeySchedule (expanded Key) for processing in the specific rounds.

Module: invmixcolumns
Inputs:[31:0] in,
Outputs:  [31:0] out
Description: Performs the mixcolumns, like its software implementation but in hardware using combinational logic.
Purpose: This module was provided and it accepts inputs, thus performing inverse mixcolumns for each 32-bit input chunk at a time. Its final implementation is when it accepts all 4 chunks of the 32-bit inputs to perform the task on the entire 128-bit message.

Module: HexDriver
Inputs: [3:0] in
Outputs: [6:0] out
Description: Displays Hexes and the proper value
Purpose: Used to display AES_Export_Data, specifically what the value of the encryption and decryption is in hexadecimal.

Module: InvSubBytes
Inputs: clk,  [7:0] in,
Output: [7:0] out
Description: Verilog implementation of InverseSubBytes
Purpose: Provided module. Used to perform the InverseSubBytes whenever called by the AES Decryption algorithm.

## Annotated (Unexpected Outputs):



## Code for TestBench

```
module testbench();

timeunit 10ns;

timeprecision 1ns;

logic CLK;
logic RESET;
logic AES_START; //initially AES_START is = 1
logic AES_DONE;  //
logic [127:0] AES_KEY;
logic [127:0] AES_MSG_ENC;
logic [127:0] AES_MSG_DEC;

AES testAES(.*);

always begin: CLOCK_GENERATION
#1 CLK = ~CLK;
end

initial begin: CLOCK_INITIALIZATION
        CLK = 0;
end

initial begin: TEST_VECTORS

RESET = 0;
AES_START = 1;
AES_KEY = 12303;
AES_MSG_ENC = 00001;



#2 RESET = 1;
#2 RESET = 0;



end
endmodule
```

**Post Lab Questions:**

| | |
|---|---|
| **LUT** | **5792** |
| **DSP** | **N/A** |
| **BRAM** | **117888** |
| **Flip-Flop** | **2343** |
| **Frequency** | 53.33 MHZ |
| **Static-Power** | **102.5 mW** |
| **Dynamic-Power** | **68.56 mW** |
| **Total-Power** | **242 mW** |

❑ **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)**

We would expect the software encryption to perform faster as compared to the hardware implementation of the decryption as the hardware performs plenty of combinational logic (such as the creation of all 10 keys from the AddRoundKey and then passing it to the system), passing through plenty of gates (thus taking us back to the conclusions made in Experiment 1) and also accesses many registers, thus it is highly dependent on the clock provided to it. As our system gets stuck in a loop, the benchmark could not be calculated, however the above explanation is our prediction based on logical reasoning and intuition.

❑ **If you wanted to speed up the hardware, what would you do? (Note: restrictions of this lab do not apply to answer this question)**

If we wanted to speed up the hardware, extending the processing rate would be a great place to start. Since the FPGA can only handle 32-bits at any given point in time, we can only perform calculations in chunks. Instead, if we moved to a 128-bit system (which would be expensive) it will certainly make the process instantaneous, almost zero latency.

**Conclusion**

There were plenty of speed bumps in this experiment and it was definitely the hardest of them all. Probably the biggest challenge we faced was grasping how the hardware and software were communicating through the avalon bus. It was an entirely new concept and it did not make sense how it was simpler than other implementations. Another major challenge was understanding how the decryption process worked because there were no tutorials given for it. It was mentioned how it is exactly identical to the encryption process but in reverse, however we were cynical in terms of how similar it might end up being. The division of bits was very confusing because we did not understand why the data types kept changing from char-pointer to int-pointer and vice versa. In all documentation it was also mentioned how the data is processed in column major order, however in reality it is actually both stored (in array) and processed in row major order and it is in fact simply printed on the screen in the column major order. This created huge problems for us as our entire algorithm seemed correct, yet the final output was incorrect.

Moreover, we messed up in the checkpoint of week 1 because we missed out on one tiny detail of shifting the bits when performing the MixColumns, thus giving expected outputs on the ModelSim, however, throwing garbage on the console when we were expecting the encrypted message. It was a quick fix which we figured out soon thereafter. In week 2, our system got stuck in a loop, thus asking us to give more attention to the state transitions thereby fixing it.

In conclusion, even though we had enough time, we were still short. Our main takeaway is to start as early as possible. The experiment was not logically challenging, difficult or requiring us to do any intense thinking, it was simply meticulous and required a lot of attention to detail. This is not the lab one should procrastinate, it is a lot of work and there is no easy way through it.