ECE385 Experiment #9

Eric Meyers, Ryan Helsdingen

Section ABG; TAs: Ben Delay, Shuo Liu

April 13th, 2016

emeyer7, helsdin2

I. Introduction

The purpose of this lab was to explore encryption/decryption techniques using the Advanced Encryption Standard (AES). The team wrote both encryption and decryption algorithms that runs on software and hardware respectively. The advantages/disadvantage of these two techniques will be analyzed in the post-lab section.

II. Description of Circuit

The encryption algorithm will be performed on a NIOS-II processor and programmed in C, whereas the decryption algorithm will be performed on a Cyclone IV FPGA and programmed in System Verilog.

The C program onboard the NIOS-II processor will communicate with the hardware on the Altera DE2-115 board and transfer both the encrypted message and key to the hardware. The hardware will then proceed to decrypt this message using the provided key and display the result on the hex display.

III. Purpose of Modules

The AES encryption/decryption algorithm used several modules. However, these can be broken down into software modules and hardware modules. The only software module was the NIOS system used to create the software encryption algorithm in C. The following were the hardware modules and these will be explained in detail when necessary:

- lab9 (top-level)
- aes_controller
- AES
- io_module
- KeyExpansion
- AddRoundKey
- SubBytes, InvSubBytes
- InvSubBytes_16
- ShiftRows, InvShiftRows
- MixColumns, InvMixColumns
- HexDriver

First, the encryption algorithm, as stated before was programmed in C on a NIOS-II processor. This processor is contained within the "NIOS system" module which is not described in this report due to repetition from last report.

The NIOS system developed was similar to that of the one created in lab8 except there are 4 PIO modules to allow communication to/from the hardware (to_sw_sig, to_hw_sig, to_sw_port, to_hw_port). A JTAG UART peripheral was used to allow debugging upon the host computer and to allow user input to the C program.

The C program was tested with input in a terminal so that the algorithm can be ensured to be correct then imported to Eclipse/Qsys to complete the software side of the lab. All functions described in the lab manual were implemented and the software successfully transmitted its contents over to the hardware to decrypt. The hardware was the trickiest part of this lab because it contained all decryption hardware modules.

lab9 (top-level)

This is the top-level module containing the connections from the nios_system, aes_controller, io_module, and the hex display.

aes_controller

The aes_controller is the module that controls the operation of the AES module. It determines if the AES decryption algorithm is in a IDLE state, COMPUTE state, or READY state. This is accomplished through a simple state machine that was provided. The only changes the team made is instead of relying on a

counter to determine when the algorithm was done, this was instead done inside the algorithm itself and output through the module into the controller.

AES

```
module  AES ( input  [127:0]  Plaintext,
                              Cipherkey,
              input           clk,
                              Reset,
                              Run,
              output logic [127:0]  Ciphertext,
              output          Ready  );
```

The AES module is the main decryption algorithm implementation. It contains inputs of Reset and Run, along with 128-bits of encrypted text and 128-bits of a key. It outputs a 128-bit value containing the plaintext of the encrypted input after finishing. The main algorithm is explained in the lab manual, as well as below in the State Diagram section.

The module essentially loops through 10 rounds of calculations with each round containing the correct cycle of calculations as defined by the state machine. In order to avoid confusion, the rounds are looped backward from Round 9 to Round 0 in order to clearly show the decryption.

io_module

```
module io_module (  input                   clk,
                    input                   reset_n,
                    output logic [1:0]      to_sw_sig,
                    output logic [7:0]      to_sw_port,
                    input [1:0]             to_hw_sig,
                    input [7:0]             to_hw_port,
                    output logic [127:0]    msg_en,
                    output logic [127:0]    key,
                    input [127:0]           msg_de,
                    output logic            io_ready,
                    input                   aes_ready,
                    output logic[6:0]       state_leds
                    );
```

The io_module provides the communication between the hardware and software. The module follows a very large state machine diagram shown in Figure 1. Inputs include the decrypted message, hardware port and sig signals to control the movement of the message, clock, reset, and AES ready bit. Outputs include 128-bit encrypted messaage and key, software port and sig signals,and an IO ready bit.

Io_module must generate the proper acknowledgement signals to send to the software and process the correct acknowledgement signals from the software. As mentioned before, the module must utilize a state-machine to generate these proper bits and this is shown later on in the ”State Diagram Section”.

KeyExpansion

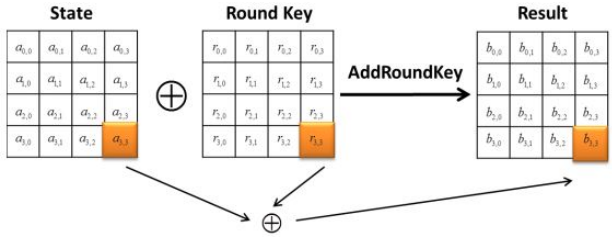
```
module KeyExpansion (input clk,
                                    input   [0:127] Cipherkey,
                                    output [0:1407] KeySchedule );
```

The KeyExpansion module takes in the Cipher Key to create the first round key and then loops 10 more times creating 128-bit Round Keys each based off the previous key. The 11 Round Keys are then stored into a 1408-bit Key Schedule.

AddRoundKey

```
module AddRoundKey(input logic [0:127] in, key,
                   output logic [0:127] out);
```

The AddRoundKey module fetches a 4-word, 128-bit Round key from the pre-computed Key Schedule and bitwise XORs with the corresponding byte from the updating 128-bit state. The figure below shows a diagram of the AddRoundKey module.

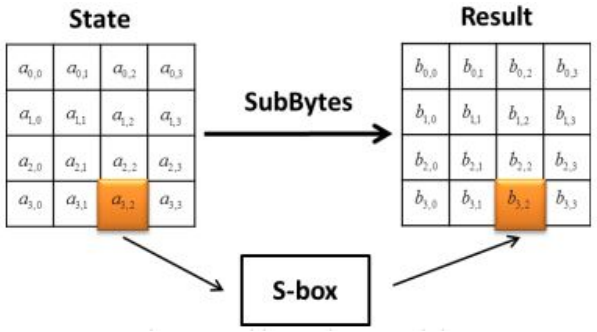


SubBytes,InvSubBytes

```
module SubBytes ( input              clk,
                  input        [0:7] in ,
                  output logic [0:7] out );
```

SubBytes takes each byte of the updating state and transforms it by taking the multiplicative inverse of the Rijindael’s finite field. The transformation has been simplified into a black-box designated as the S-box (substitution box). Inputs include the 256 byte array and clock. Outputs include the transformed 256 byte array. The figure below portrays this module as a 4x4 matrix for sample.

The InvSubBytes module is similar to the SubBytes module except that the S-box is replaced by its inverse



to reverse the encryption process.

InvSubBytes_16

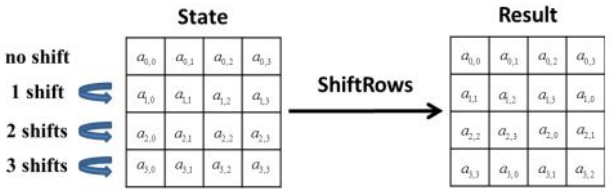
```
module InvSubBytes_16(input logic clk,
                      input logic [0:127]  in,
                      output logic [0:127] out);
```

InvSubBytes_16 is a submodule of InvSubBytes. It takes in the entire state during the InvSubBytes round and performs the appropriate substitution for each row and column.

ShiftRows,InvShiftRows

```
module InvShiftRows(input logic [0:127] in, output logic[0:127] out);
```

The ShiftRows module takes in the 128-bit state and shifts the elements in each row n of the matrix by n-1 to the left. The figure below shows a diagram of ShiftRows.



The InvShiftRows module does exactly what The ShiftRows module does except it reverses the direction such the row n is right-circularly shifted by n-1 times.

MixColumns,InvMixColumns

The MixColumns module takes each of the four words from the state individually and goes through invertible linear tranformations over GF(2^8) including multiplication by a fixed polynomial matrix to linearly combine the four bytes of each word to form a new Word.

HexDriver

```
module InvMixColumns ( input        [0:31] in ,
                       output logic [0:31] out );
```

This module contains the logic necessary to output values in binary/decimal in hex on the hex displays on board the Altera DE2-115.

IV. Software/Hardware State Diagram

The software/hardware state diagram is handled in io_module and is shown in Figure 1.

V. Decryption State Diagram

The decryption state diagram is split up into two FSMs for simplicity. One is referred to as the Calculation FSM and the other is referred to as the Round FSM. There are a total of 10 rounds with an extra round to ensure the key_expansion algorithm performs. Each round a series of calculations are performed as determined in the algorithm outlined in the description PDF of this lab. The Round FSM is shown in Figure 2 and the Calculation FSM is shown in Figure 3. Last, the AES Controller state machine is shown in Figure 7.

VI. Schematic/Block Diagram

The schematics and block diagram for the Top Level Module, the AES Controller, and AES Encryption Module can be found in ”Figures Section”.

VII. Annotated Pre-Lab Waveforms

The annotated pre-lab waveforms can be found in Figure 8, 9, 10, and 11 for a 2000ns period. The ending decrypted value given plaintext of ”daec3055df058e1c39e814ea76f6747e” with a key of ”0102030405060708090a0b0c0d0e0f” is ”ece298ece298...dc”.

VIII. Post Lab

1) Which would you expect to be faster to complete encryption/decryption, the software or hardware? Is this what your results show?
   - The faster encryption/decryption is expected to be the hardware because programming in software (specifically in C) is meant for generalized processors and thus must go through more clock cycles to do more compared to that of the hardware. The hardware is dedicated to the task at hand, and thus is faster. Specifically about 10 times as fast as the software as shown below by our

Resource	Value
LUT	5510
DSP	0
Memory (BRAM)	599,040
Flip-Flop	667
Frequency	141.88 MHz
Static Power	102.29 mW
Dynamic Power	0.77 mW
Total Power	187.27 mW

TABLE I: Design Statistics

metrics.

Software Speed: 0.0058 MB/s
Hardware Speed: 0.00043 MB/s (simulation)

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)
   - Instead of containing separate modules for each of the operations, these can be combined into a single module and the entire computation for one round can be done in a single state. This would significantly decrease the amount of time taken for one state to process.

IX. Conclusion

Overall, this lab proved to be very difficult from both a hardware and software perspective. Programming the encryption algorithm in C was difficult for many reasons. Working with multidimensional arrays is never an easy task in C, and this lab seemed to put this skill to the test. The notation used throughout the algorithm is difficult to understand fully what should be passed into certain functions. However, once a basic working model of the encryption algorithm was developed, it was optimized for patterns, and condensed and therefore much easier to understand.

The hardware decryption was just as difficult as the software, if not more difficult. The finite state machines that were developed took a large amount of time to debug using ModelSim. The io_module used to pass information to/from the software proved to be a large time-sink in and of itself.

In the end, the team did not receive full points for demo due to a glitch in passing the data to/from software. However, partial credit was received for showing simulations.

Overall this lab was one of the most difficult ECE385 labs, however it was one of the most rewarding to understand in the end. AES encryption/decryption is now very clear to the team.

X. Figures

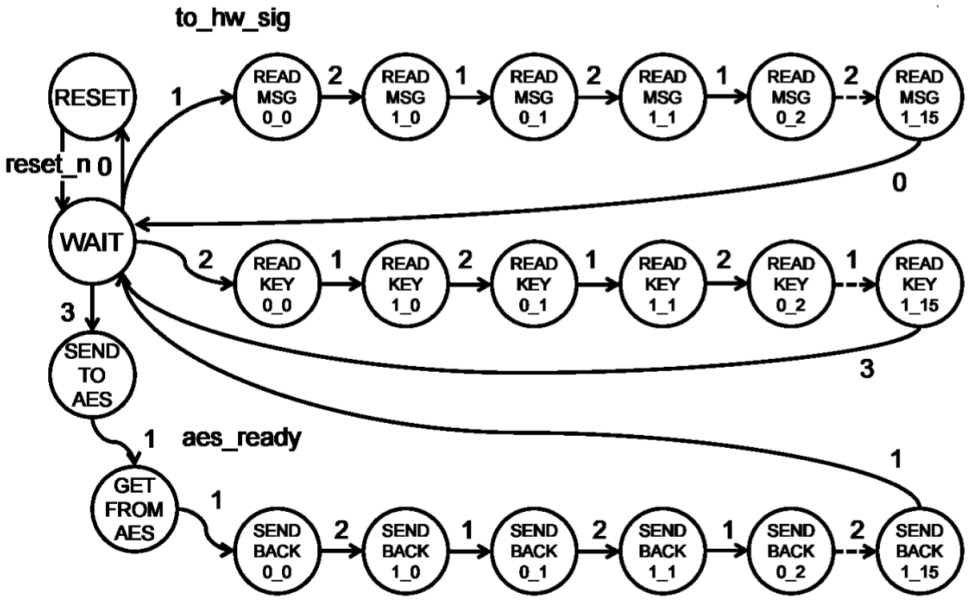


Fig. 1: IO Module Hardware/Software State Diagram

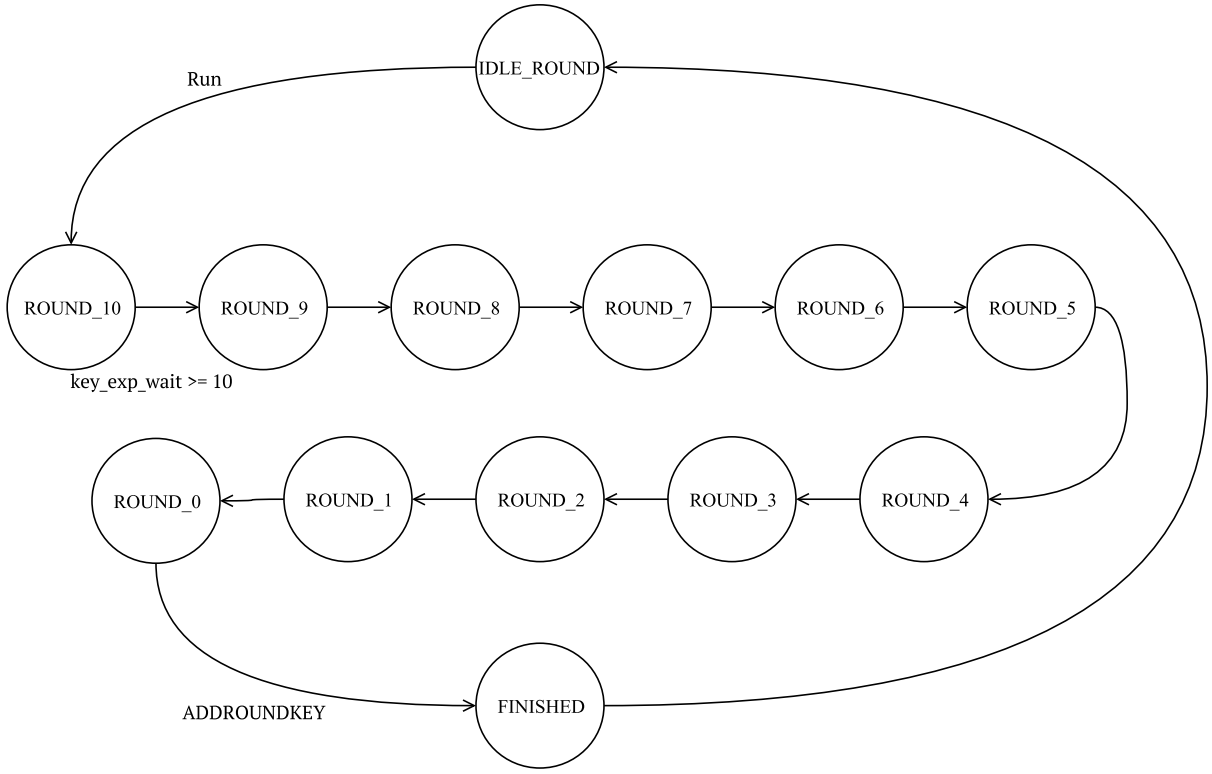


Fig. 2: Decryption Round State Diagram

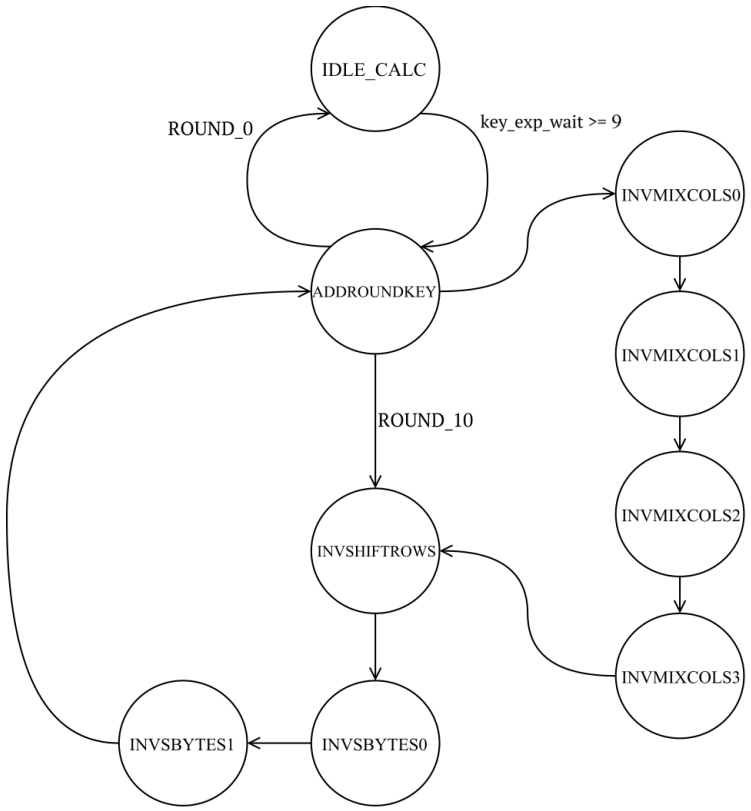


Fig. 3: Calculation State Diagram

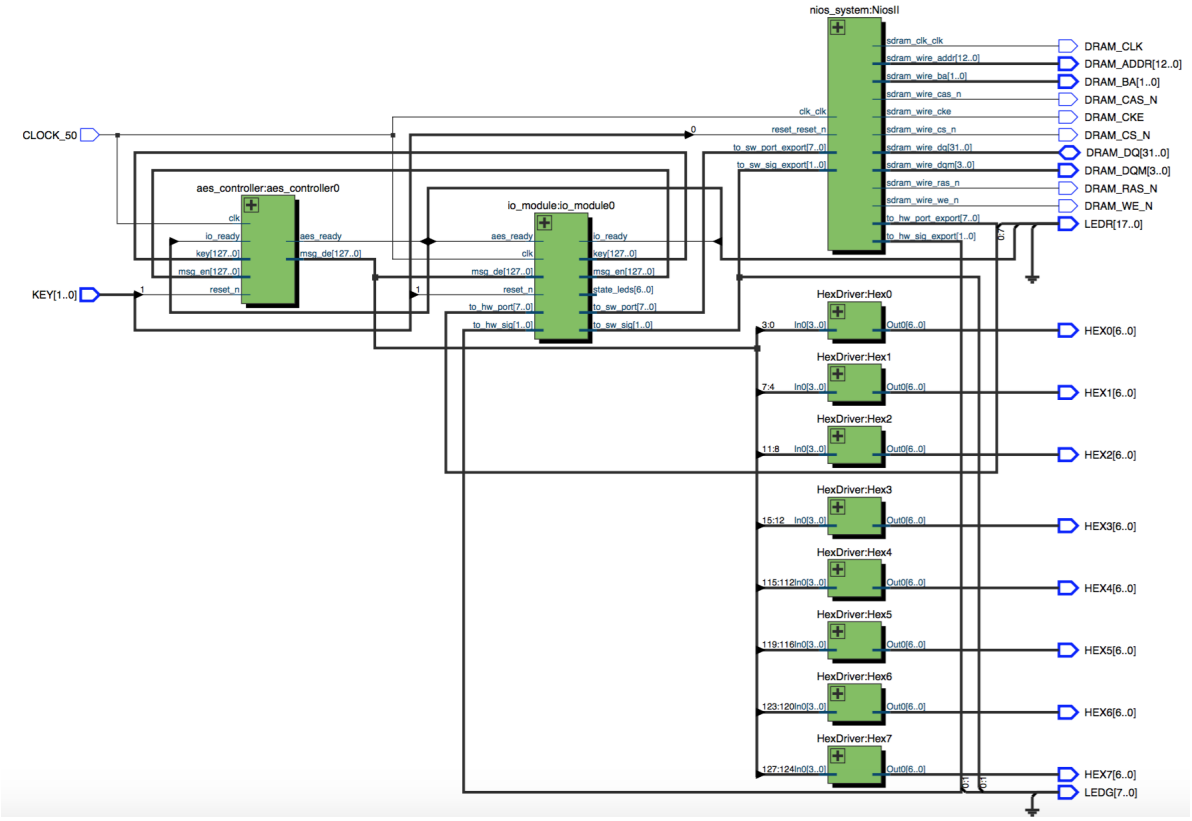


Fig. 4: Top Level Circuit Diagram

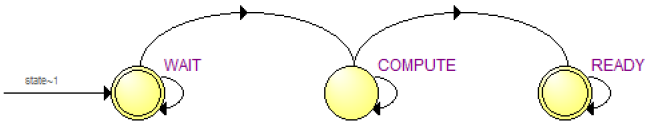


Fig. 5: AES Controller State Diagram

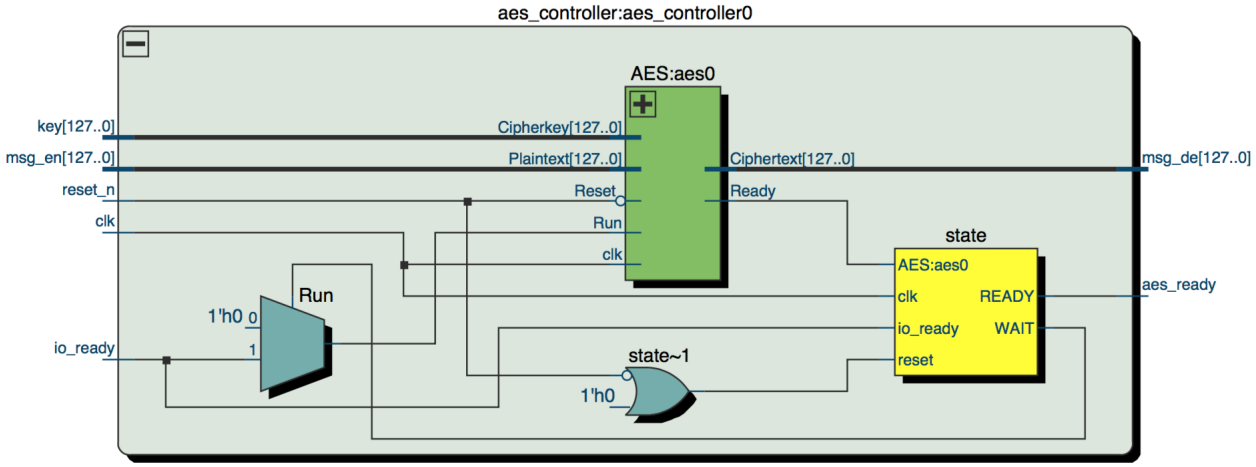


Fig. 6: AES Controller Circuit

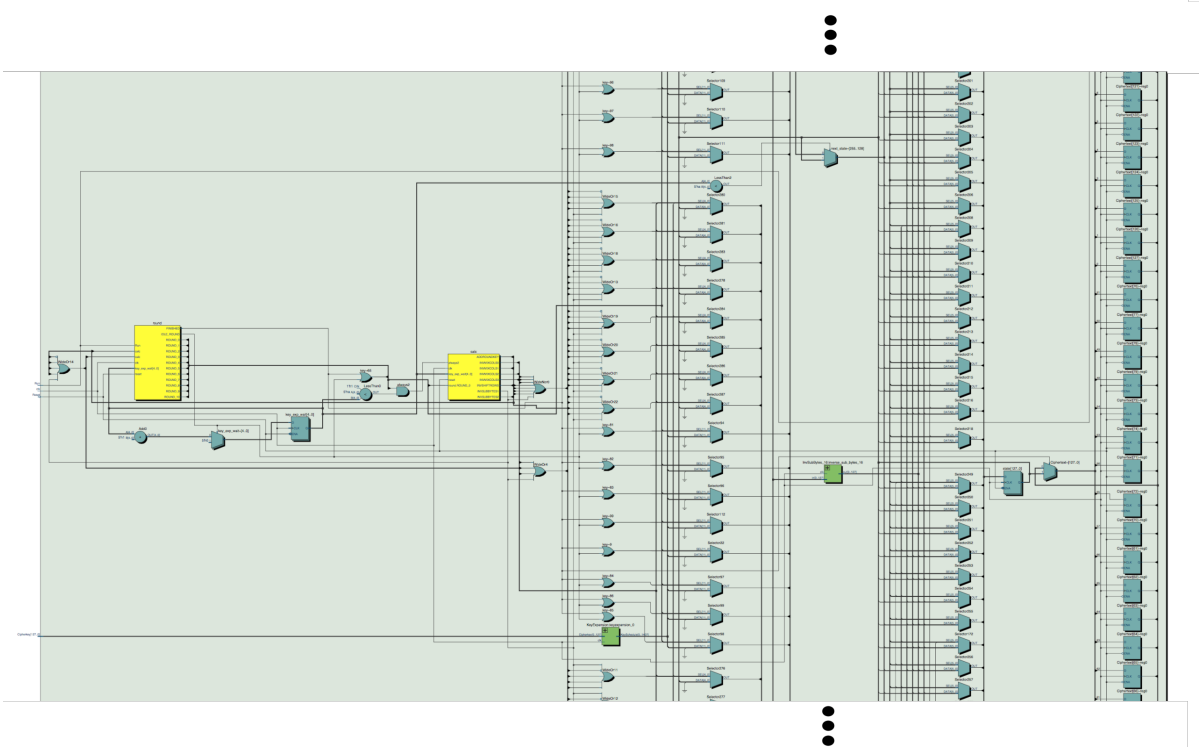


Fig. 7: AES Circuit

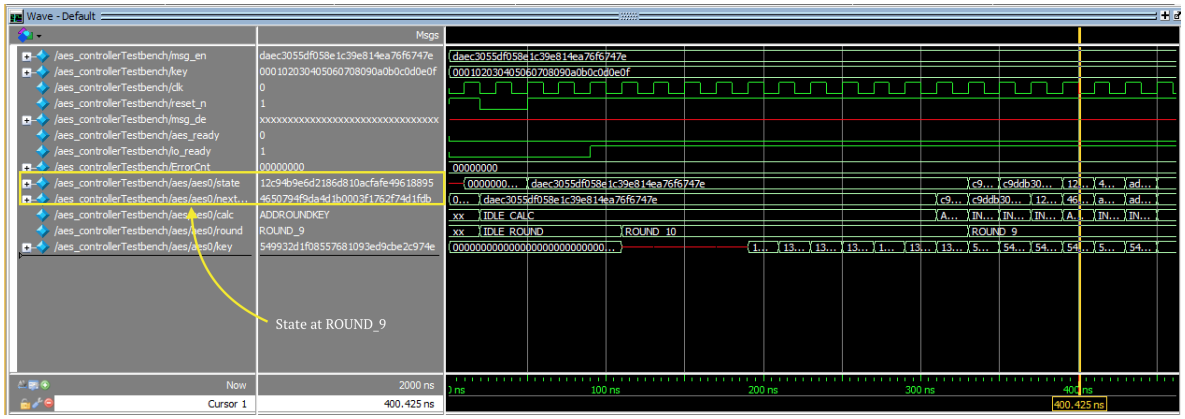


Fig. 8: Simulation 0 - 500ns

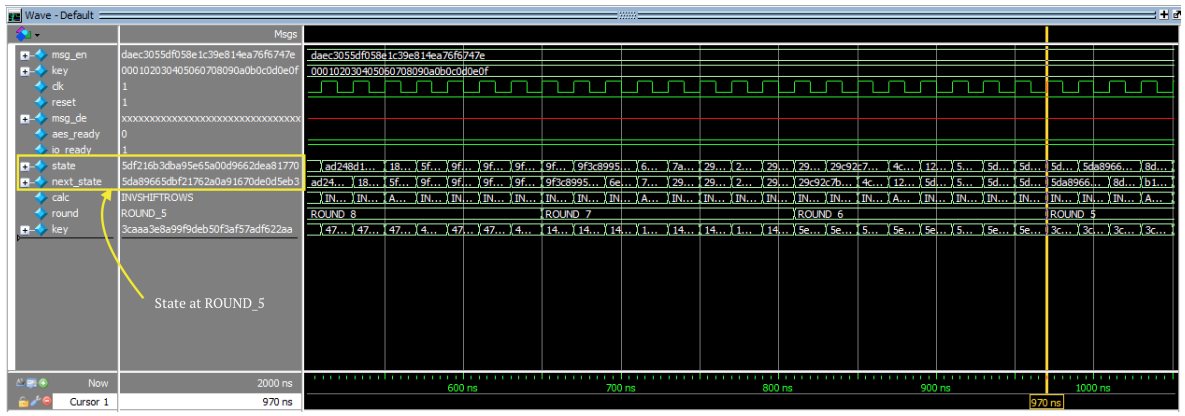


Fig. 9: Simulation 500 - 1000ns

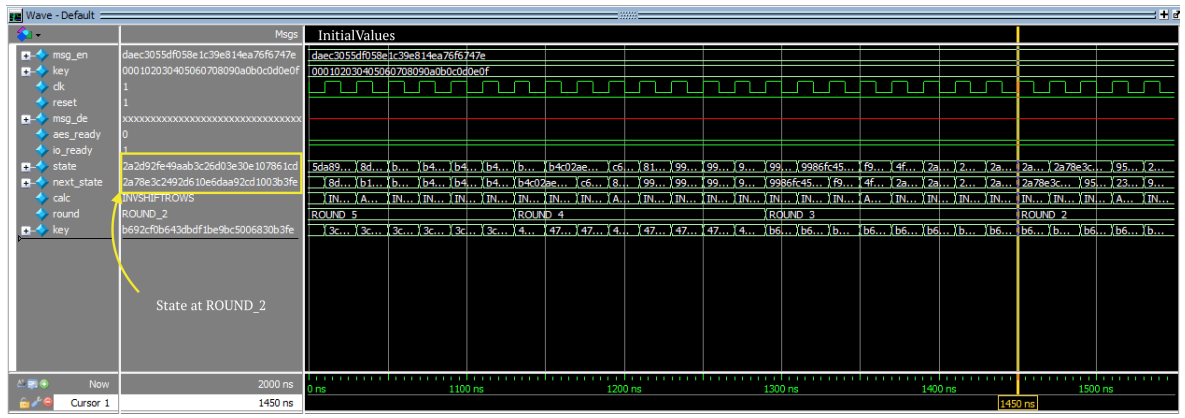


Fig. 10: Simulation 1000 - 1500ns

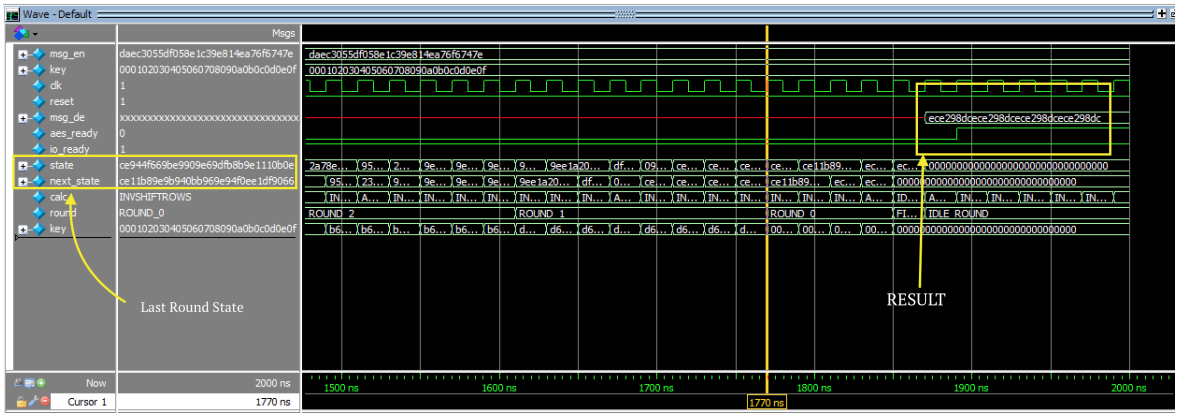


Fig. 11: Simulation 1500-2000ns