ECE 385

Spring 2020

Experiment #9

SOC with Advanced Encryption Standard (AES) in System Verilog

Freddy Zhang and Mehul Dugar

Section: ABO (Thu. 3PM)

Gene Shiue

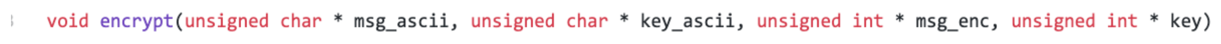
Introduction:

In this week’s lab, we were tasked to implement a 128-bit AES (Advanced Encryption Standard) using SystemVerilog as an IP (Intellectual Property) core. This lab was split into two weeks, in week 1 we implemented the software side (Encryption) with the help of the software IP core (in Eclipse) and the Platform Designer. Week 2 involved getting the decryption algorithm working in SystemVerilog using modules and using Platform Designer to implement the hardware IP core. To sum the lab up, our objective was to create a hardware/software interface that was capable of implementing the AES, in other words, capable of encrypting and decrypting the message with a specific key.

Description of AES (Encryptor/Decryptor):

Encryption:

This functionality was implemented on the software side of the lab. The interface for the encryption involved the user being prompted to enter a message to be encrypted along with a key to perform the AES with. The input message and the key are 32 characters each (these characters represent hex values). Our test case used the input message: “daec3055df058e1c39e814ea76f6747e” and our key was “000102030405060708090a0b0c0d0e0f”. This message and key is passed to the software side of things and are pointed at by the msg_ascii and key_ascii pointers, which are parameters to the encrypt function.



```
void encrypt(unsigned char * msg_ascii, unsigned char * key_ascii, unsigned int * msg_enc, unsigned int * key)
```

In the function *encrypt*, the first task is to convert the input key and message into hexadecimal values so we can manipulate the data, this was accomplished by using the *charsToHex* function. Then we followed the algorithm that was given to us in the lab manual. We first called the *KeyExpansion* function to generate a key schedule (of size 176 chars) based on the input key. The remaining functions were called as per the algorithm given in the figure below.

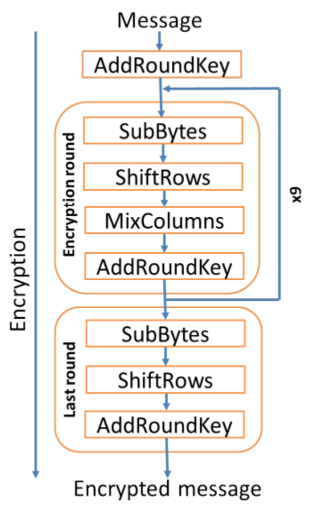


Figure 2 AES encryption algorithm flow.

At the end of the encrypt function, in order to pass on the results to the hardware (Just the Hex Display for week 1) we assigned the AES_PTR to the key so that they would be displayed on the Hex Display. AES_PTR essentially copied the key values and stored them in the registers in avalon_aes_interface.

<u>Decryption:</u>

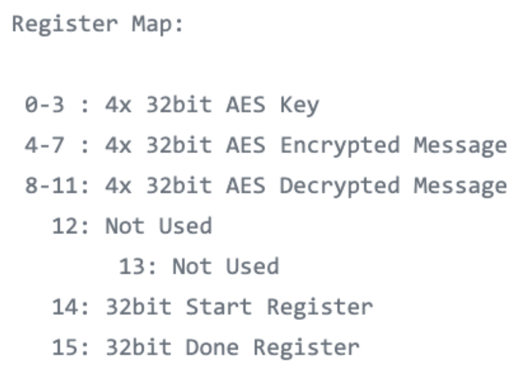
We implemented the decryption aspect using the hardware component of this lab. More specifically it was implemented in the AES.sv file along with the AES_Control.sv (state machine).

```
module AES (
        input    logic CLK,
        input  logic RESET,
        input  logic AES_START,
        output logic AES_DONE,
        input  logic [127:0] AES_KEY,
        input  logic [127:0] AES_MSG_ENC,
        output logic [127:0] AES_MSG_DEC
);
```

In this sv file, we first instantiate the state machine module. Then we create a KeyExpansion module followed by a AddRoundKey module. These modules were followed by InvShiftRows and InvMixColumns modules. We also had to make 4 modules of the InvSubHelper (A module that calls InvSubBytes 4 times). This consisted of all the modules called in this .sv file. We decided to use two 2-bit signals from the state machine to assist with performing the appropriate operation as well as choosing the appropriate input. Op_out was used to decide between the 4 functions in the loop and wd_out was used to choose between 8 byte words as inputs for the InvMixColumns function.

Hardware-Software Interface:

The interface between the hardware and the software was implemented in the avalon_aes_interface.sv.



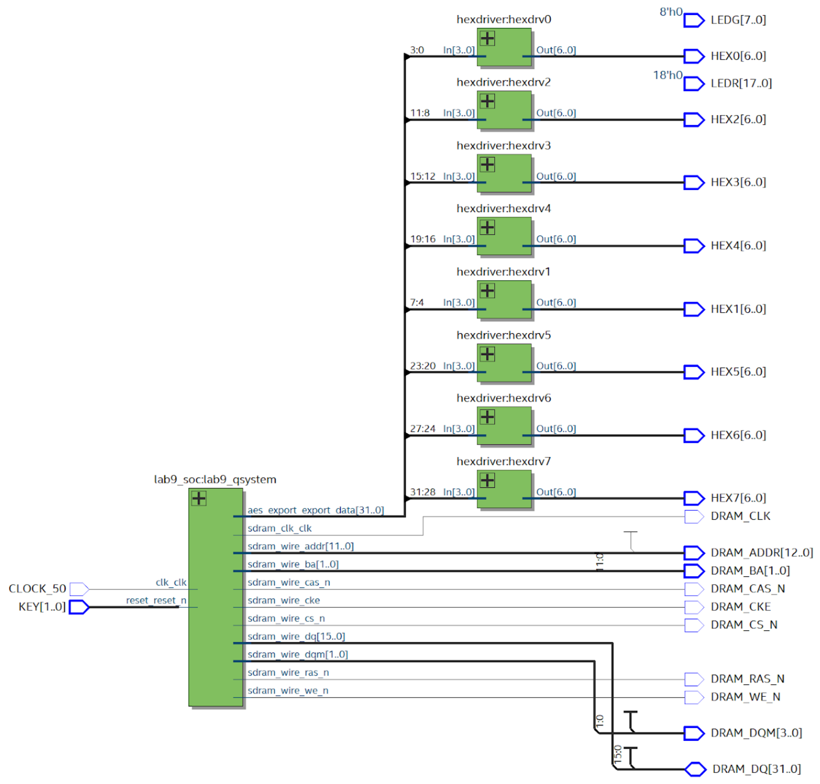
```
Register Map:

 0-3 : 4x 32bit AES Key
 4-7 : 4x 32bit AES Encrypted Message
 8-11: 4x 32bit AES Decrypted Message
   12: Not Used
        13: Not Used
   14: 32bit Start Register
   15: 32bit Done Register
```

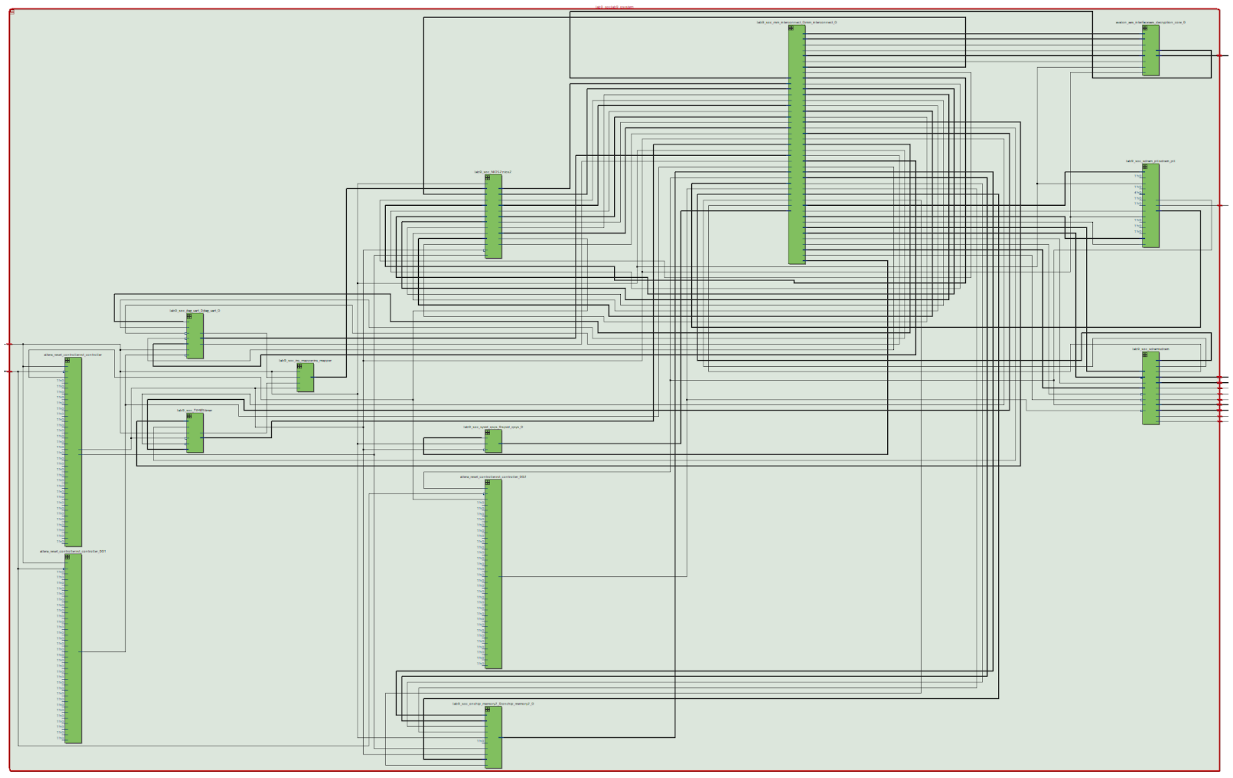
Here we instantiate the AES module and then update the registers based on the outputs. Apart from this, we also have other input logic such as CS (chip select), READ, WRITE and BYTEENABLE which help in performing the read and write operations by the Avalon-slave module. The decryption process continues till AES_DONE = 1, which is updated by the AES module.

Block Diagram

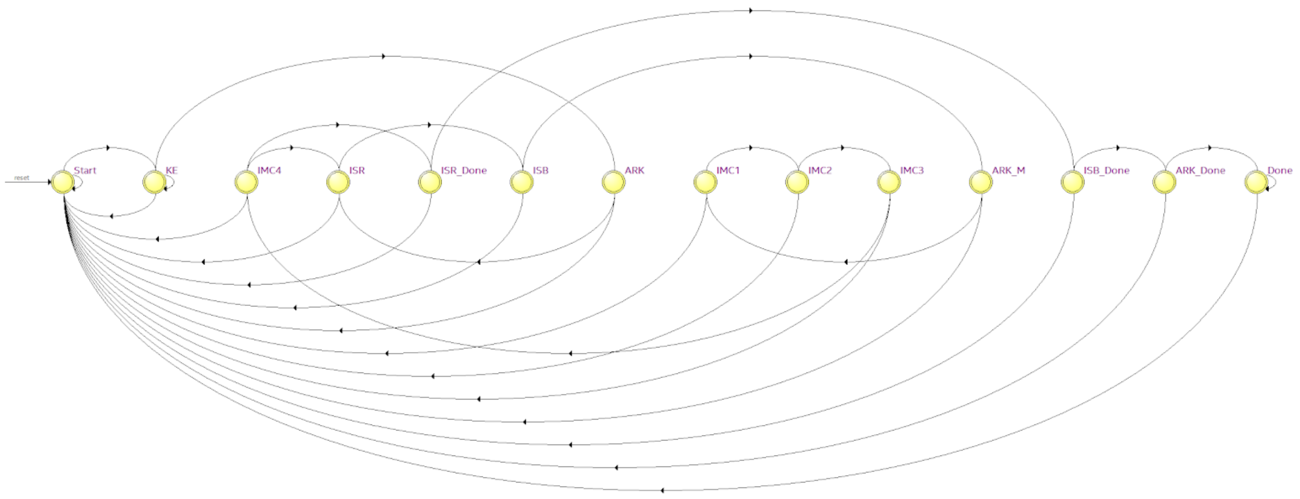
Top-level



avalon_aes_interface.sv



State Diagram:



Module Descriptions:

.sv Files:

Module: lab9_top

Input: CLOCK_50, [1:0] KEY

Output: [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, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK

Inout: [31:0] DRAM_DQ

Description: This module is responsible for instantiating all other modules as it is the toplevel

Purpose: Build interface between NIOS II and output for Hexdrivers

Module: avalon_aes_interface.sv

Input: CLK, RESET, AVL_READ, AVL_WRITE, AVL_CS, [3:0] AVL_BYTE_EN, [3:0] AVL_ADDR, [31:0] AVL_WRITREDATA

Output: [31:0] AVL_READDATA

Description: This module is the decryption core that stores the data necessary to interact with the software in the NIOS II and the hardware on the board.

Purpose: IP core necessary for software to hardware communication

Module: InvSubBytes

Input: clk, [7:0] in

Output: [7:0] out

Description: performs the InvSubBytes AES decryption step

Purpose: One of the steps necessary for the AES decryption algorithm

Module: KeyExpansion.sv

Input: clk, [127:0] Cipherkey

Output: [1407:0] KeySchedule

Description: This performs the KeyExpansion operation that provides round keys for AES decryption and encryption

Purpose: This is used to perform the AddRoundKey module

Module: InvShiftRows.sv

Input: [127:0] data_in

Output:[127:0] data_out

Description: This performs the InvShiftRows operation that shifts the rows of the input by a certain based on the AES decryption algorithm

Purpose: This is part of the AES Decryption algorithm

Module InvMixColumns.sv

Input:[31:0] in

Output: [31:0] out

Description: This multiplies the input matrix by a specific matrix to invert the MixColumns operation done by the AES encryption algorithm

Purpose: This is part of the AES Decryption algorithm to decode the encrypted message

Module: hexdriver.sv

Input: [3:0] In

Output: [6:0] out

Description: Converts a hex value to be displayed on the board

Purpose: used to display the key on the board and test the code

Module: AES.sv

Input: CLK, RESET, AES_START, [127:0] AES_KEY, [127:0] AES_MSG_ENC, [127:0] AES_MSG_DEC

Output: AES_DONE

Description: This module will take an encrypted message as input and output the decrypted message

Purpose: This is the central unit that contains all of the modules necessary to decrypt a message in one place.

Module: AES_Control.sv

Input: CLK, RESET, AES_START

Output: AES_DONE, [4:0] CTR, [1:0] op_out, [1:0] wd_out

Description: State controller of the entire AES Decryption algorithm

Purpose: This controls which operations to do in order to successfully decrypt the input message

Module: AddRoundKey.sv

Input: CLK, [127:0] msg, [10:0] key[127:0], [4:0] round

Output: [127:0] out

Description: This module adds the round key to the current message. The round key is created by KeyExpansion.sv and the round is determined by AES_Control

Purpose: This is part of the AES decryption algorithm

Module: InvSubHelper.sv

Input: clk, [31:0] in

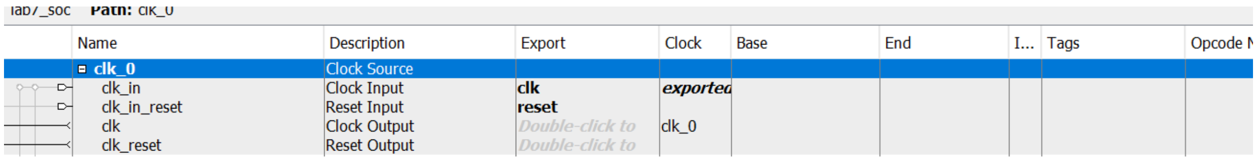
Output: [31:0] out

Description: This is the helper function of InvSubBytes so that it is more easier to handle and more manageable

Purpose: This makes it easier to code and it also makes the code in AES.sv more condensed which makes it easier to read.

<u>Platform Designer:</u>

clk_0

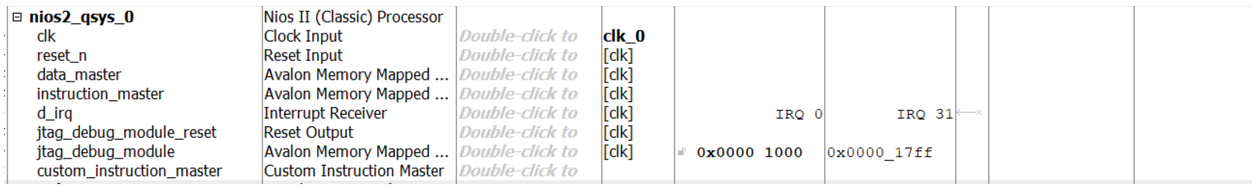


lab7_soc Path: clk_0

Name	Description	Export	Clock	Base	End	I...	Tags	Opcode N
clk_0	Clock Source							
clk_in	Clock Input	**clk**	*exported*					
clk_in_reset	Reset Input	**reset**						
clk	Clock Output	*Double-click to*	clk_0					
clk_reset	Reset Output	*Double-click to*						

This is the clock module that enforces the 50MHz generated by the FPGA.

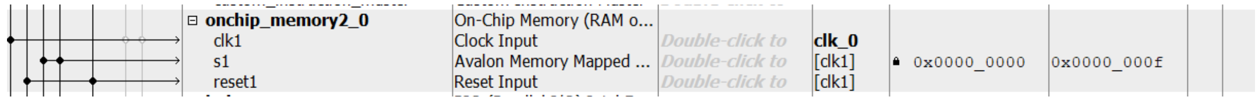
nios2_qsys_0



nios2_qsys_0	Nios II (Classic) Processor					
clk	Clock Input	*Double-click to*	**clk_0**			
reset_n	Reset Input	*Double-click to*	[clk]			
data_master	Avalon Memory Mapped ...	*Double-click to*	[clk]			
instruction_master	Avalon Memory Mapped ...	*Double-click to*	[clk]			
d_irq	Interrupt Receiver	*Double-click to*	[clk]	IRQ 0	IRQ 31	
jtag_debug_module_reset	Reset Output	*Double-click to*	[clk]			
jtag_debug_module	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0000 1000	0x0000_17ff	
custom_instruction_master	Custom Instruction Master	*Double-click to*				

This module is the NIOS II/e processor. This module is responsible for performing all processing on instructions.

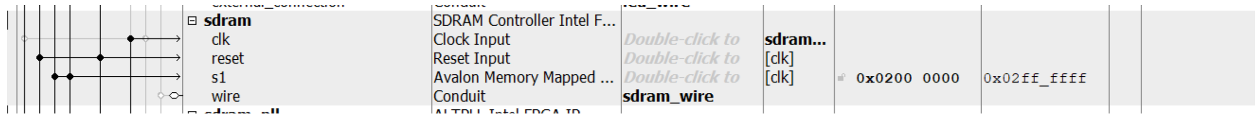
onchip_memory2_0



⊟ onchip_memory2_0	On-Chip Memory (RAM o...				
clk1	Clock Input	Double-click to	clk_0		
s1	Avalon Memory Mapped ...	Double-click to	[clk1]	0x0000_0000	0x0000_000f
reset1	Reset Input	Double-click to	[clk1]		

As the name suggests, this is the on chip memory. It is smaller than the SRAM but faster. It has 32 bit width and 16 bytes memory size

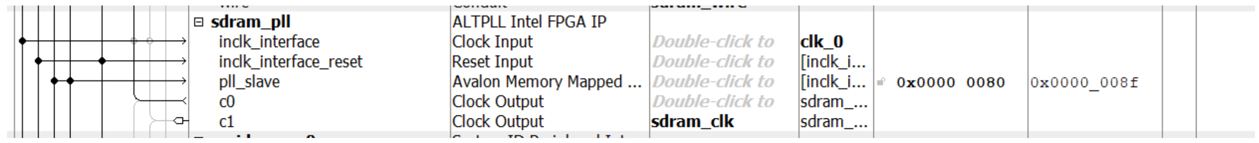
sdram



⊟ sdram	SDRAM Controller Intel F...				
clk	Clock Input	Double-click to	sdram...		
reset	Reset Input	Double-click to	[clk]		
s1	Avalon Memory Mapped ...	Double-click to	[clk]	0x0200 0000	0x02ff_ffff
wire	Conduit	sdram_wire			

This is the module used to implement the SDRAM, this is where the system program is stored as there is limited storage in onchip memory.

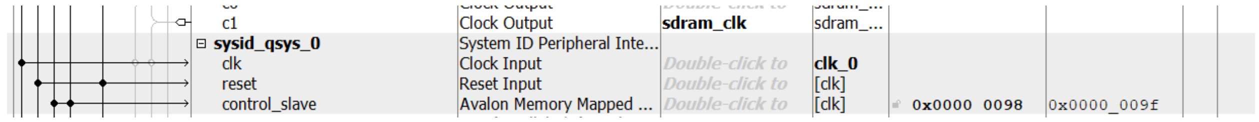
sdram_pll



⊟ sdram_pll	ALTPLL Intel FPGA IP				
inclk_interface	Clock Input	Double-click to	clk_0		
inclk_interface_reset	Reset Input	Double-click to	[inclk_i...		
pll_slave	Avalon Memory Mapped ...	Double-click to	[inclk_i...	0x0000 0080	0x0000_008f
c0	Clock Output	Double-click to	sdram_...		
c1	Clock Output	sdram_clk	sdram_...		

This module is the clock for the SDRAM, it helps in rectifying timing errors by accounting for delays

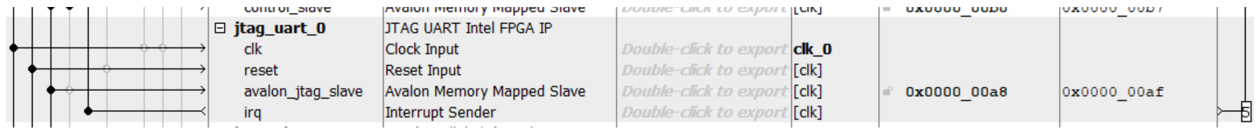
sysid_qsys_0



c1	Clock Output	sdram_clk	sdram_...		
⊟ sysid_qsys_0	System ID Peripheral Inte...				
clk	Clock Input	Double-click to	clk_0		
reset	Reset Input	Double-click to	[clk]		
control_slave	Avalon Memory Mapped ...	Double-click to	[clk]	0x0000 0098	0x0000_009f

This is the system ID checker which was introduced in the tutorial. It helps in avoiding any incompatibility issues between the hardware and the software.

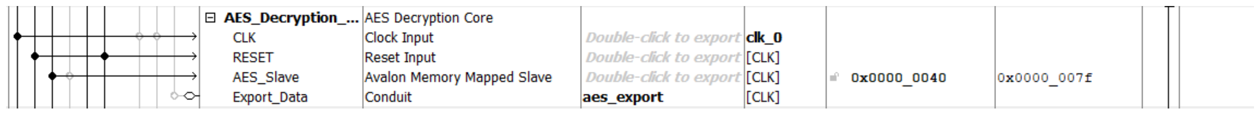
jtag_uart_0



⊟ jtag_uart_0	JTAG UART Intel FPGA IP				
clk	Clock Input	Double-click to export	clk_0		
reset	Reset Input	Double-click to export	[clk]		
avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_00a8	0x0000_00af
irq	Interrupt Sender	Double-click to export	[clk]		

The JTAG UART allows the host computer to communicate with NIOS II so that the NIOS II can receive the keypresses.

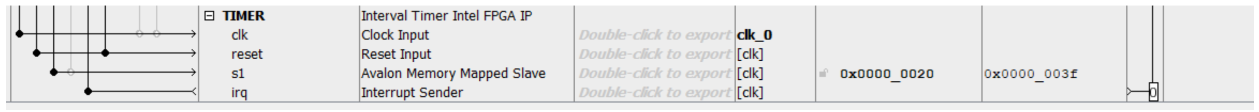
AES_Decryption_core



⊟ AES_Decryption_...	AES Decryption Core				
CLK	Clock Input	Double-click to export	clk_0		
RESET	Reset Input	Double-click to export	[CLK]		
AES_Slave	Avalon Memory Mapped Slave	Double-click to export	[CLK]	0x0000_0040	0x0000_007f
Export_Data	Conduit	aes_export	[CLK]		

The AES_Decryption_Core allows an IP to be created to implement a module that can communicate with the hardware and software.

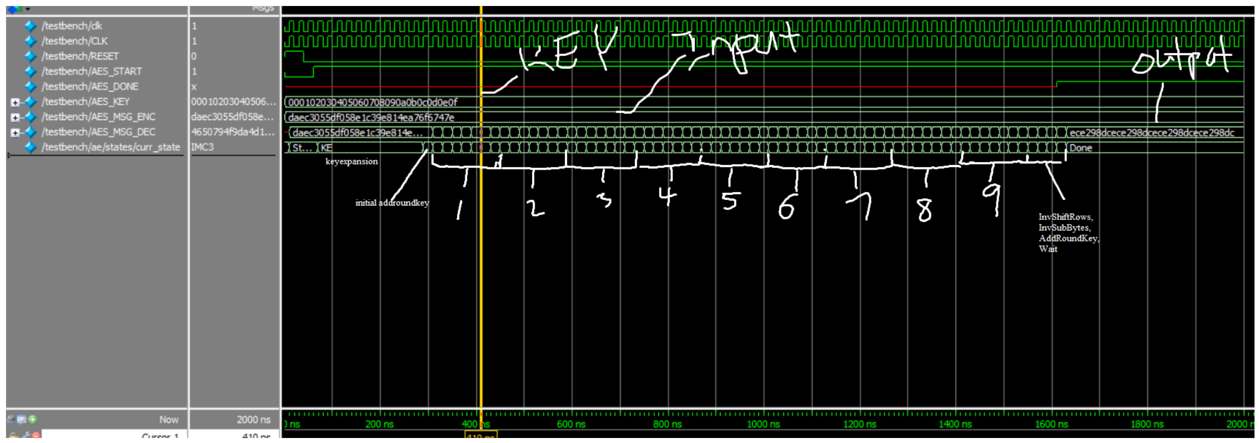
Timer



⊟ TIMER	Interval Timer Intel FPGA IP				
clk	Clock Input	Double-click to export	clk_0		
reset	Reset Input	Double-click to export	[clk]		
s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0020	0x0000_003f
irq	Interrupt Sender	Double-click to export	[clk]		

The timer was used to accurately get the speed of the software encryption and hardware decryption.

Annotated Simulations:



/testbench/clk	1
/testbench/CLK	1
/testbench/RESET	0
/testbench/AES_START	1
/testbench/AES_DONE	x
/testbench/AES_KEY	000102030405060...
/testbench/AES_MSG_ENC	daec3055df058e...
/testbench/AES_MSG_DEC	4650794f9da4d1...
/testbench/ae/states/curr_state	IMC3

KEY Input
output
000102030405060708090a0b0c0d0e0f
daec3055df058e1c39e814ea76f6747e
daec3055df058e1c39e814e...
ece298dcece298dcece298dcece298dc
St...
KE
Done
keyexpansion
initial addroundkey
1
2
3
4
5
6
7
8
9
InvShiftRows, InvSubBytes, AddRoundKey, Wait
0 ns
200 ns
400 ns
600 ns
800 ns
1000 ns
1200 ns
1400 ns
1600 ns
1800 ns
2000 ns
Now
2000 ns
Cursor 1
410 ns

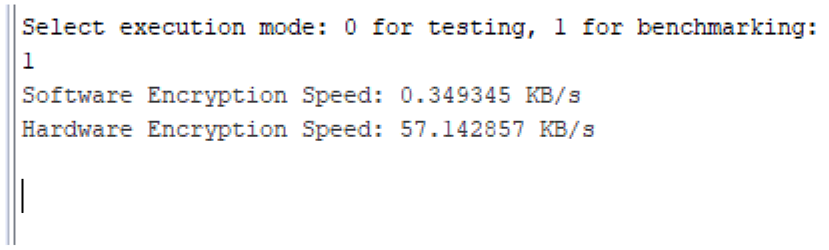
Post Lab Questions:

Design Resources and Statistics Table :

LUT	**6389**
DSP	**0**
BRAM	**126080**
Flip-Flop	**2828**
Frequency	**70.67 MHz**
Static Power	**102.26 mW**
Dynamic Power	**0.75 mW**
Total Power	**175.29 mW**

a. 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 encryption/decryption to be faster on the hardware. Our results were:



```
Select execution mode: 0 for testing, 1 for benchmarking:
1
Software Encryption Speed: 0.349345 KB/s
Hardware Encryption Speed: 57.142857 KB/s
```

Since NIOS II is not a dedicated processor, rather, it is synthesized on the FPGA, and that the FPGA also needs to be able to store the hardware modules we create, we are restricted in regards to size of the processor. Thus performance is sacrificed due to lack of space available.

Additionally, taking into consideration the execution process of hardware modules vs. NIOS II, NIOS II has to fetch and decode each instruction before it can start computation.

This is in contrast to the hardware modules which do not require the fetch or decode states. This definitely helps the hardware modules to be faster.

b. If you wanted to speed up the hardware, what would you do? (Note: Restrictions of this lab do not apply to this question)

   I think the slowest part of the hardware was the InvMixColumns module, it only operated on 8 bytes at one time. If we were able to parallelly perform InvMixColumns on the entire message, the process would be much faster. In the current implementation, 8 bytes are processed every clock cycle.

Conclusions:

Our design was partially functional on the demo day, we were unable to get an accurate output to the board but the simulation worked without hiccups. The software was successfully able to encrypt the given message and the hardware was successfully able to decrypt the message. In the first week, the Hex Displays displayed the first 2 and last 2 bytes of the key sent. This was done successfully. In the second week we had to implement the Decrypt algorithm, and we had to display the first 4 and the last 4 characters of the decrypted message. We were able to decrypt the message in the simulation waveform but were unable to translate it onto the Hex Displays. We believe that the issue lies in storing the decrypted message in the registers. Since the waveform had the correct output, we inferred that the problem had to come from being able to store the data in the registers correctly for Eclipse to read as plain text.

One thing that could be improved could be better instructions for how to use SignalTap as we felt a little inexperienced while trying to use it.