ECE 385

Fall 2019

Experiment #9

SOC with Advanced Encryption Standard in SystemVerilog

Xinglong Sun Churan He
ABD T 8AM
Mihir Iyer

Introduction

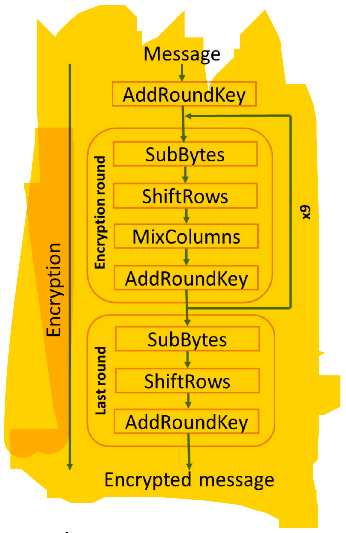
a. Briefly summarize the operation of the AES encryptor/decryptor.

In Lab 9, we design and implement 128-bit AES (Advanced Encryption Standard) encryptor and decryptor on NIOS II SOC. In the encryption process, the 128-bit Plaintext could be encrypted with a 128-bit Cipher Key to form a 128-bit Ciphertext. The encryption process contains the initial "AddRoundKey", nine rounds of "SubBytes, ShiftRows, MixColumns, and AddRoundKey", and the last round "SubBytes, ShiftRows, AddRoundKey". We implement this functions with C program running in NIOS II CPU. As for the decryption process, the 128-bit Ciphertext can be decrypted with the Cipher Key and obtain the original Plaintext. The decryption is the reverse process of the encryption, which contains the initial "AddRoundKey", nine rounds of "InvShiftRows, InvSubBytes, AddRoundKey, and InvMixColumns", and the last round "InvShiftRows, InvSubBytes, AddRoundKey". We implement the decryption functionality purely with hardware. With the benchmark feature in the C program, we are able to benchmark our encryptor and decryptor in order to show the advantage in computing speed of hardware implementation.

Written Description and Diagrams of the AES encryptor/decryptor

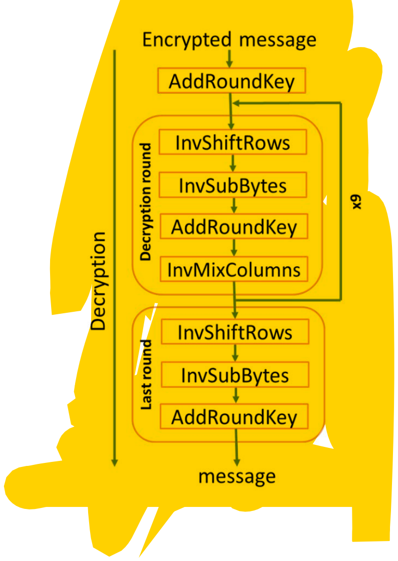
a. Written description of the software encryptor

In this lab, we do the Encryption part in NIOS II processor. The 128-bit AES Encryption algorithm is written in C and allows user to input the plain text as message and cipher key to the Eclipse Console. More concretely, we break the encryption algorithm into five subroutines which are: KeyExpansion, AddRoundKey, SubState, ShiftRows, and MixColumns. The order and number of times we implement each step is specified in the flowchart attached below. Basically, KeyExpansion will pre-compute every round key that could be used in the AddRoundKey phase and store them in a key-matrix. This step is carried out only once at the beginning of each encryption. AddRoundKey will perform an addition of the key of the current round and the current state message in GF(2^8) domain. SubState will transform and replace each byte of the current state with a SubByte using an 8-bit substitution box. ShiftRows will perform (n-1) circular left shifts on nth row of our current state. MixColumns will multiply each column by a fixed constant polynomial matrix.



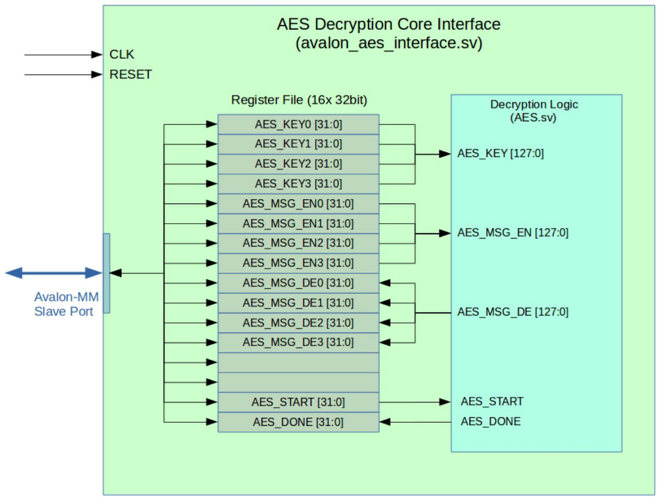
b. Written description of the hardware decryptor

In this lab, we implement the decryption part purely with hardware logics. At the beginning of the decryption, the C program write the Ciphertext and the Cipher Key to specified locations in the register file of the AES Decryption Core Interface Module. At the same time, the register that contains START become 1 and the register contains DONE become 0. These data become the input of the AES module, which is the computation hardware module to decrypt the Ciphertext. The decryption process contains the initial “AddRoundKey”, nine rounds of “InvShiftRows, InvSubBytes, AddRoundKey, and InvMixColumns”, and the last round “InvShiftRows, InvSubBytes, AddRoundKey”. Thus we need to design a moore state machine to process the data accordingly. Since the initial KeyExpansion and AddRoundKey takes 10 clock cycles, and there are 10 more AddRoundKeys, We include a “count to 10” counter in our state machine to simplify the design. The moore state machine controls the select signal of the MUX connected to different computation module, and provides the round data for AddRoundKey module. As the decryption process is finished. The AES module will output the Plaintext to the interface, which would in turn read by the C program.



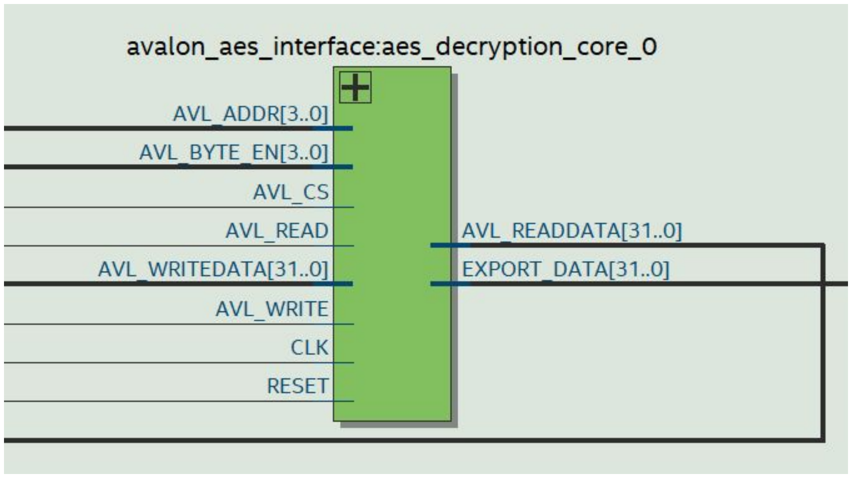
c. Written description of the hardware/software interface (avalon_aes_interface.sv)

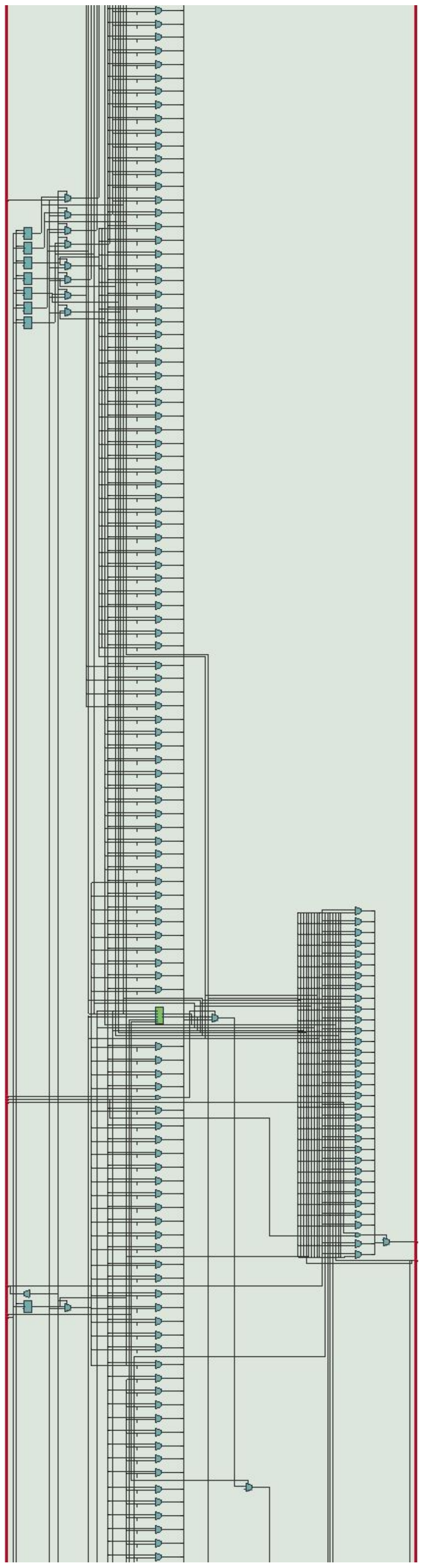
The communication of NIOS II software and hardware is implemented in the avalon_aes_interface module. Basically, we created a register file with 16 32-bit registers that cache the values used by our Hardware Decryption and Software Encryption. How we come up with design of the reg file is straightforward. Since we need to store Encrypted Message, Key, and Decrypted Message, and each of them is 128-bit long, we need 4 32-bit registers to store each of them, yielding a total of 12 registers. We need two more registers to hold the START and DONE signals that indicate when to start decryption and when the decryption is done. These two signals ensure that our hardware part can work seamlessly with the software and perform consecutive actions. These two additional registers brings the total to 14. We add two more blank registers to make the total number 16(2 ^ 4). The detail of each register is illustrated in the diagram attached below. Since our AES Decryption Core (with top-level avalon_aes_interface.sv) is also memory-mapped on the Avalon Bus, we can directly access these registers by dereferencing the pointers in NIOS II.



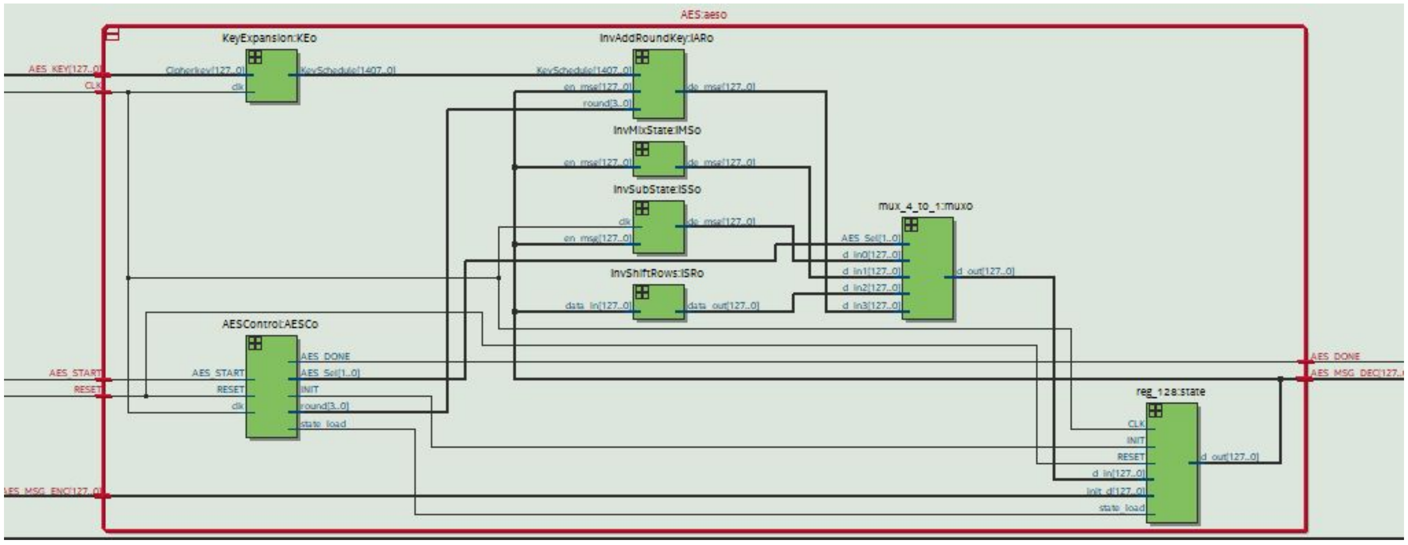
d. Block diagram(s)

I. RTL view of avalon_aes_interface.sv

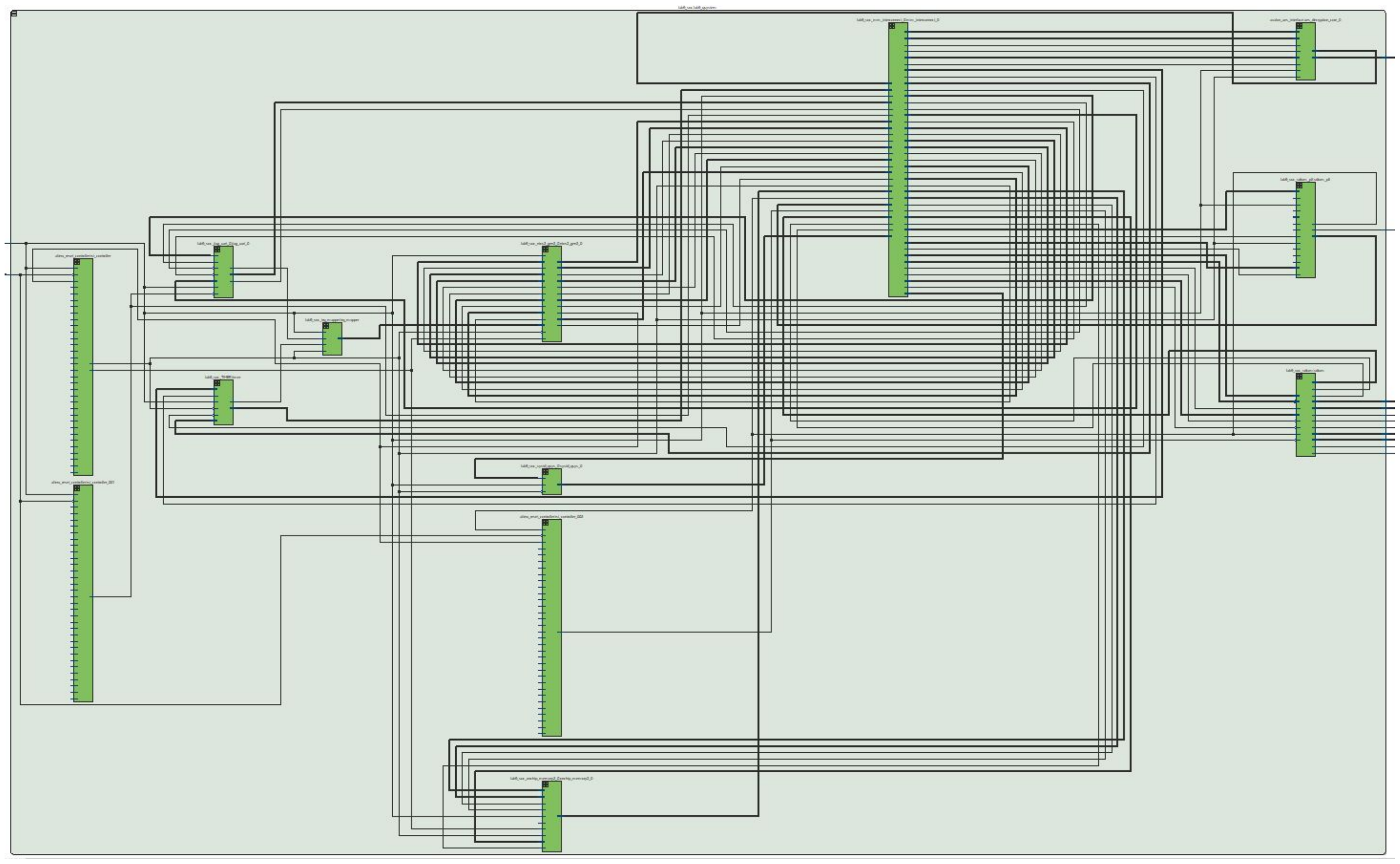




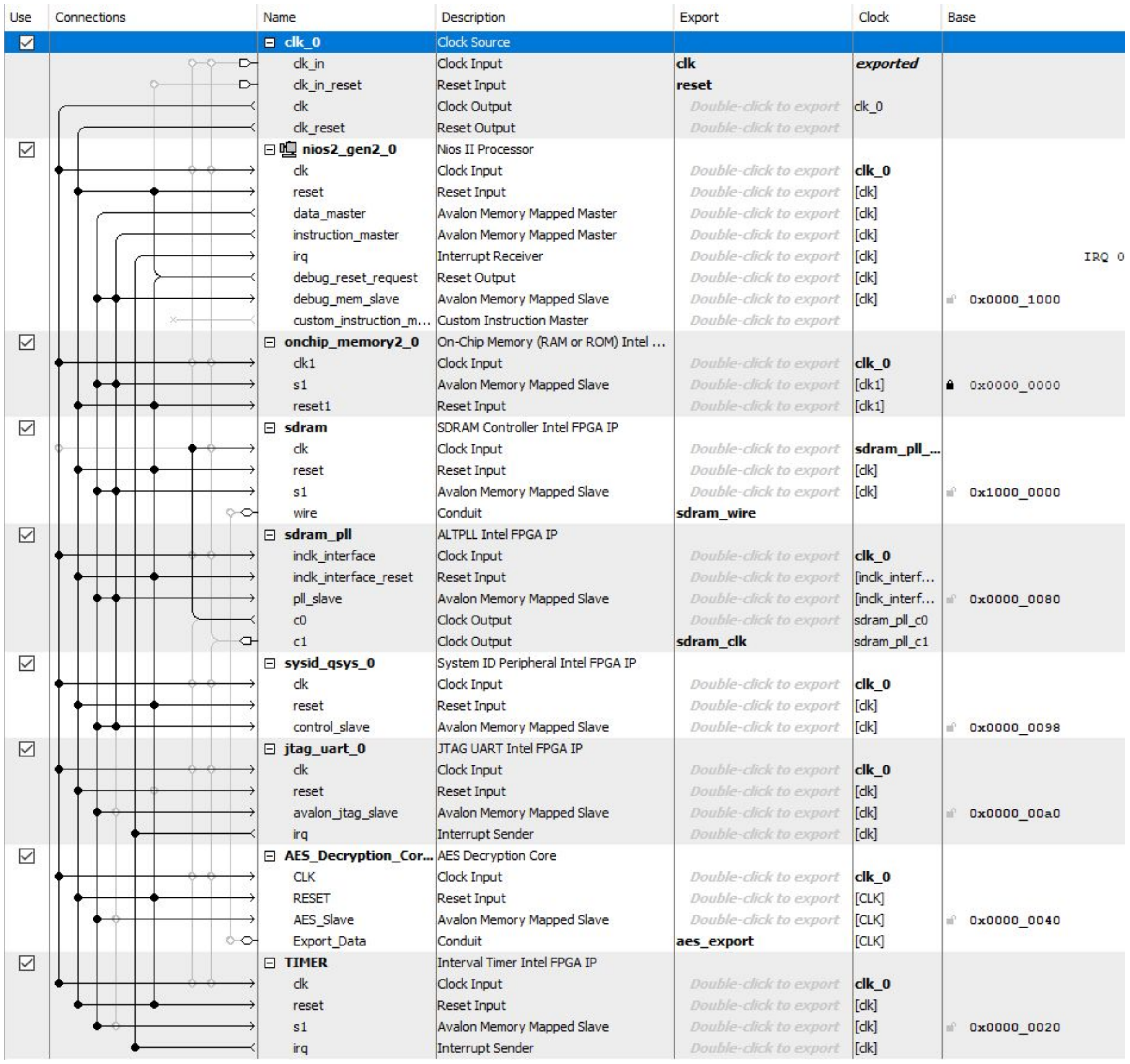
ii. AES RTL diagram



iii. Top-level RTL diagram



iv. Qsys diagram



Use	Connections	Name	Description	Export	Clock	Base	
☑		⊟ clk_0	Clock Source				
		clk_in	Clock Input	**clk**	***exported***		
		clk_in_reset	Reset Input	**reset**			
		clk	Clock Output	*Double-click to export*	clk_0		
		clk_reset	Reset Output	*Double-click to export*			
☑		⊟ nios2_gen2_0	Nios II Processor				
		clk	Clock Input	*Double-click to export*	**clk_0**		
		reset	Reset Input	*Double-click to export*	[clk]		
		data_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]		
		instruction_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]		
		irq	Interrupt Receiver	*Double-click to export*	[clk]		IRQ 0
		debug_reset_request	Reset Output	*Double-click to export*	[clk]		
		debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_1000	
		custom_instruction_m...	Custom Instruction Master	*Double-click to export*			
☑		⊟ onchip_memory2_0	On-Chip Memory (RAM or ROM) Intel ...				
		clk1	Clock Input	*Double-click to export*	**clk_0**		
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	0x0000_0000	
		reset1	Reset Input	*Double-click to export*	[clk1]		
☑		⊟ sdram	SDRAM Controller Intel FPGA IP				
		clk	Clock Input	*Double-click to export*	**sdram_pll_...**		
		reset	Reset Input	*Double-click to export*	[clk]		
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x1000_0000	
		wire	Conduit	**sdram_wire**			
☑		⊟ sdram_pll	ALTPLL Intel FPGA IP				
		inclk_interface	Clock Input	*Double-click to export*	**clk_0**		
		inclk_interface_reset	Reset Input	*Double-click to export*	[inclk_interf...		
		pll_slave	Avalon Memory Mapped Slave	*Double-click to export*	[inclk_interf...	0x0000_0080	
		c0	Clock Output	*Double-click to export*	sdram_pll_c0		
		c1	Clock Output	**sdram_clk**	sdram_pll_c1		
☑		⊟ sysid_qsys_0	System ID Peripheral Intel FPGA IP				
		clk	Clock Input	*Double-click to export*	**clk_0**		
		reset	Reset Input	*Double-click to export*	[clk]		
		control_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0098	
☑		⊟ 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_00a0	
		irq	Interrupt Sender	*Double-click to export*	[clk]		
☑		⊟ AES_Decryption_Cor...	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	
		Export_Data	Conduit	**aes_export**	[CLK]		
☑		⊟ 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	
		irq	Interrupt Sender	*Double-click to export*	[clk]		

e. State Diagram of AES decryptor controller

   i. The state machine in AES.sv.

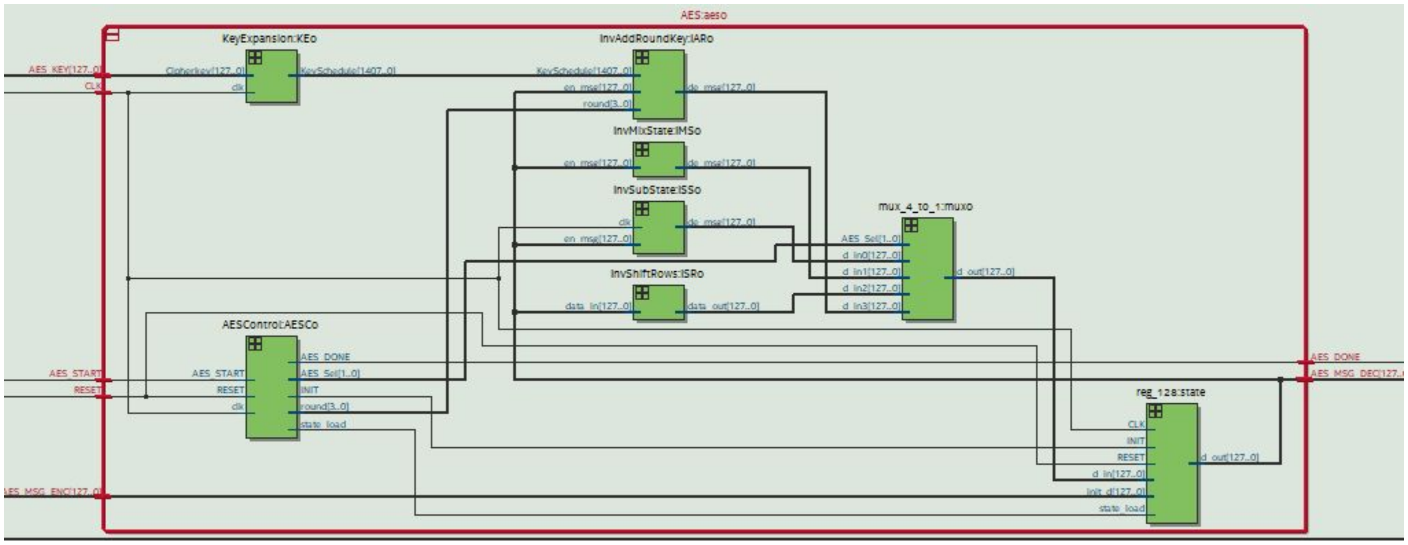
   ii. You may abbreviate the 9 looping rounds in the state diagram like in figure 9 on page IAES.9 of the lab manual.

Module Descriptions

SystemVerilog Modules:

AES.sv
Inputs:
CLK, RESET, AES_START
[127:0] AES_KEY, AES_MSG_ENC
Outputs:
AES_DONE
[127:0] AES_MSG_DEC
Purpose & Description:

This is the top level unit that wraps our AES Decryption components together. In this module, we add a 4-to-1 Mux that serves as the datapath in our project. At each clock cycle, exactly one of the outputs of the four aforementioned operations (AddRoundKey, InvMixColumns, InvSubState, InvShiftRows) is selected and loaded back to our state register. The select signals of the MUX and load enable signal on our register are both controlled by the AESControl Unit which is a Moore State Machine. Besides the 128-bit long decrypted message, the unit outputs a single-bit AES_DONE signal indicating that our decryption process is completed.



avalon_aes_interface.sv
Inputs:
CLK, RESET, AVL_READ,AVL_WRITE, AVL_CS,
[3:0] AVL_BYTE_EN, AVL_ADDR,

[31:0] AVL_WRITEDATA

Outputs:

[31:0] AVL_READDATA, EXPORT_DATA

Purpose & Description:

As elaborated in the above subsection, this module serves as the bridge between our hardware and NIOS II software. It contains a reg file that stores the encrypted message, decrypted message, cypher key, and start and end signals. AVL_ADDR ,AVL_WRITEDATA, and AVL_READDATA are the address and data that software wants to access and store. The input AVL_BYTE_EN decides what part of the specific register you want to access.

lab9_top.sv

Inputs:

CLOCK_50,
[1:0] KEY

Outputs:

[7:0] LEDG,
[17:0] LEDR,
[6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7
[12:0] DRAM_ADDR
[1:0] DRAM_BA
DRAM_CAS_N, DRAM_CKE, DRAM_CS_N, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK
[3:0] DRAM_DQM

Inout:

[31:0] DRAM_DQ

Purpose & Description:

It's the top level entity of Lab 9 hardware, and it connects the Lab 9 hardwares with off-chip clock generator, keys, LEDs, and the SDRAM.

counter.sv

Inputs:

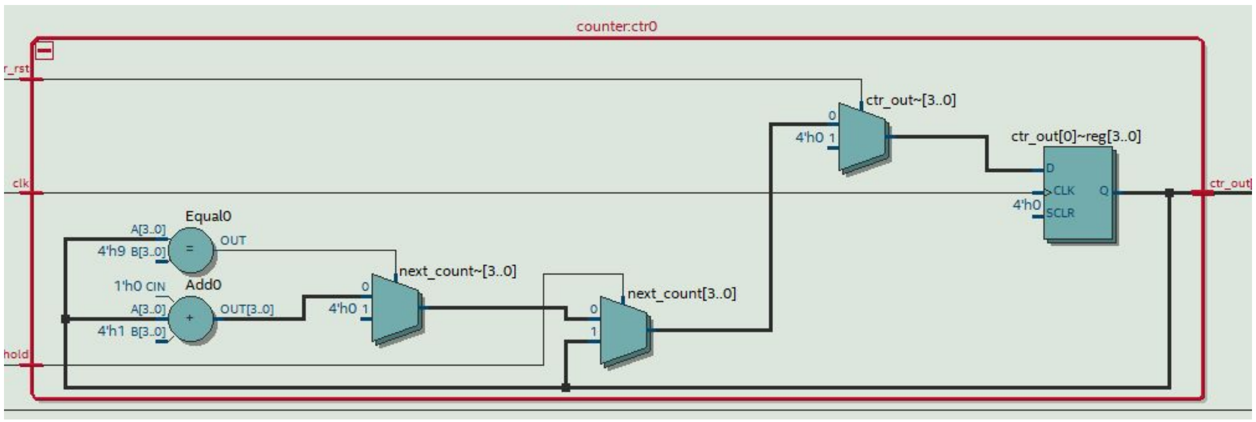
ctr_rst,
clk,
hold,

Outputs:

[3:0] ctr_out

Purpose & Description:

The counter is used in the moore state machine for the transition logic and the “round” input for AddRoundKey module. This counter would count from 0 to 9, because KeyExpension and the first AddRoundKey needs 10 clock cycles and there are other 10 AddRoundKeys. This counter has a special ‘hold’ input, so we can hold the counter’s current value in each 4-procedure round.



AESControl.sv

Inputs:

RESET, AES_START, clk

Outputs:

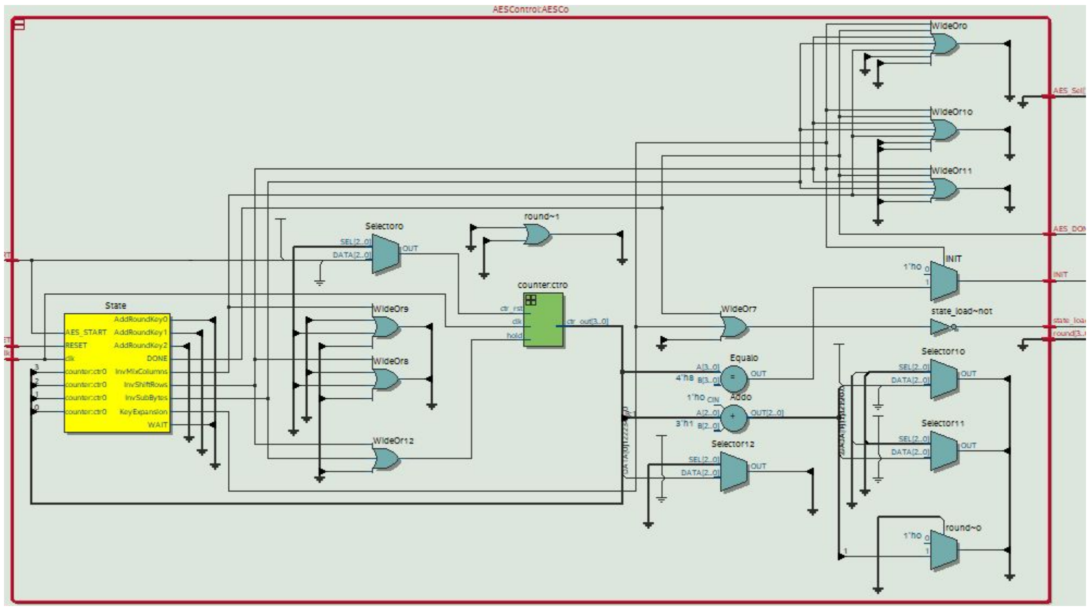
[3:0] round,

[1:0] AES_Sel,

AES_DONE, state_load, INIT

Purpose & Description:

This module is the control unit in our design. It implements a Moore FSM. With the application of counter, we shrink the total number of states to 9. Among the input signals, RESET brings our state machine back to the initial WAIT state. AES_START triggers the decryption process and moves our state machine to the first decryption step. Among the output signals, “round” indicates the number of round we’re currently at, which is used by AddRoundKey to fetch the proper key corresponding to the specific round. AES_Sel controls our 4-to-1 MUX to select the proper data to be fed back into the “state” register. State_load determines whether we want to update the data stored in our “state” register. INIT will load the “state” register with the initial message information, and AES_DONE indicates the completeness of our entire decryption process.



InvAddRoundKey.sv

Inputs:

[1407:0] KeySchedule,

[127:0] en_msg,

[3:0] round

Outputs:

[127:0] de_msg

Purpose & Description:

This modules combines our current state with a roundKey based on the current round number. The 1408 bit long KeySchedule is pre-computed and generated within the KeyExpansion phase, and it contains 11 round keys (each of 128 bit). En_msg is the current state data, and the 4-bit long "round" indicates which round we're currently at, ranging from 1 to 11. De_msg is the XOR of the current state data with the proper round key.

InvMixColumns.sv

Inputs:

[31:0] in

Outputs:

[31:0] out

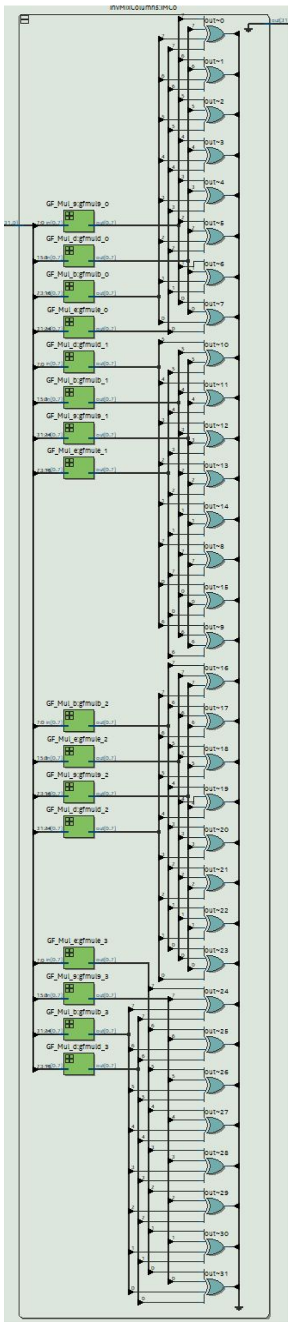
Purpose & Description:

We compute the dot product between the 32-bit input and a fixed polynomial matrix c(x) as specified below:

$$\begin{bmatrix} b_{0,i} \\ b_{1,i} \\ b_{2,i} \\ b_{3,i} \end{bmatrix} = \begin{bmatrix} a_{0,i} \\ a_{1,i} \\ a_{2,i} \\ a_{3,i} \end{bmatrix} \bullet \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix},$$

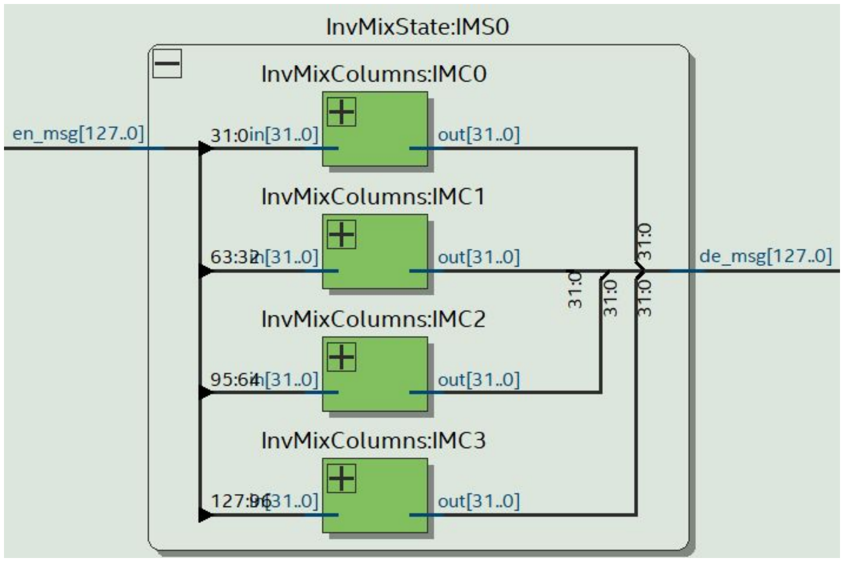
where the resulting column is calculated by:

$$\begin{aligned} b_{0,i} &= (\{0e\} \bullet a_{0,i}) \oplus (\{0b\} \bullet a_{1,i}) \oplus (\{0d\} \bullet a_{2,i}) \oplus (\{09\} \bullet a_{3,i}) \\ b_{1,i} &= (\{09\} \bullet a_{0,i}) \oplus (\{0e\} \bullet a_{1,i}) \oplus (\{0b\} \bullet a_{2,i}) \oplus (\{0d\} \bullet a_{3,i}) \\ b_{2,i} &= (\{0d\} \bullet a_{0,i}) \oplus (\{09\} \bullet a_{1,i}) \oplus (\{0e\} \bullet a_{2,i}) \oplus (\{0b\} \bullet a_{3,i}) \\ b_{3,i} &= (\{0b\} \bullet a_{0,i}) \oplus (\{0d\} \bullet a_{1,i}) \oplus (\{09\} \bullet a_{2,i}) \oplus (\{0e\} \bullet a_{3,i}) \end{aligned}.$$



InvShiftRows.sv
Inputs:
[127:0] data_in,
Outputs:
[127:0] data_out
Purpose & Description:
This module performs inversed shift rows function. Specifically, it performs n right circular shifts on nth row. Notice that we perform n left circular shifts on the nth row in the encryption process. In decryption, this process is reversed. The 128-bit data_in is the current state data, and 128-bit data_out is the data of the next state.

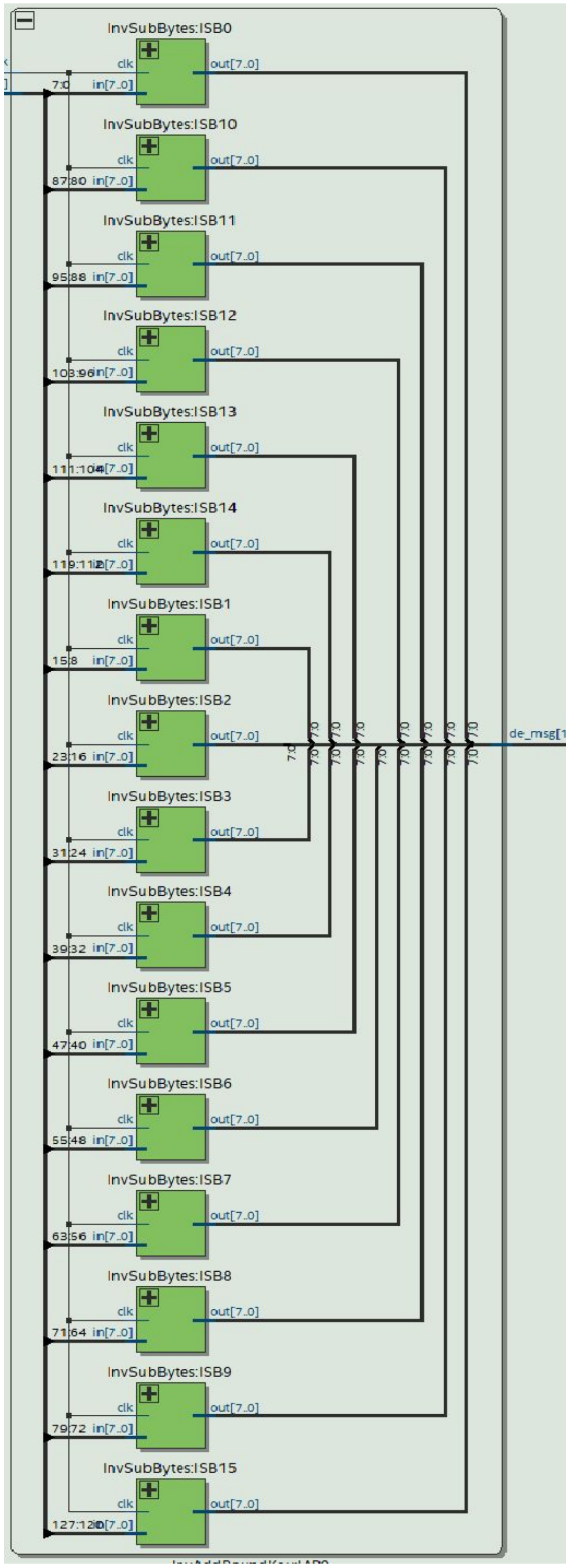
InvMixState.sv
Inputs:
[127:0] en_msg
Outputs:
[127:0] de_msg
Purpose & Description:
This module consists of four InvMixColumns described above. This is created for convenience, and it substitutes each column with the InvMixColumn output.



InvSubState.sv
Inputs:
clk,
[127:0] en_msg,
Outputs:
[127:0] de_msg

Purpose & Description:

It consists 32 InvSubBytes. InvSubState are able to transform the entire state according to the predefined mapping table.



KeyExpansion.sv

Inputs:

Clk,

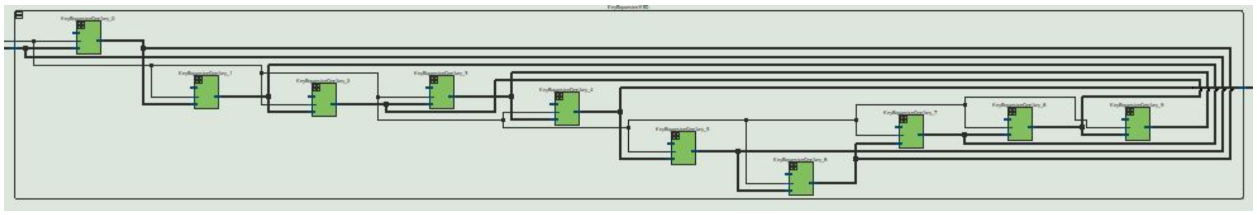
[127:0] Cipherkey

Outputs:

[1407:0] KeySchedule

Purpose & Description:

This module takes in a 128 bit long cipher key and generates a series of round keys that will be used in the AddRoundKey step. Basically, it follows the AES KeyExpansion algorithm. One thing that differentiates the hardware usage(decryption) of roundkey with the software implementation(encryption) is that the order of applying round keys to the state data is actually reversed. Concretely, the last set of round key we use in the software is used initially in the hardware.



InvSubBytes.sv

Inputs:

clk,

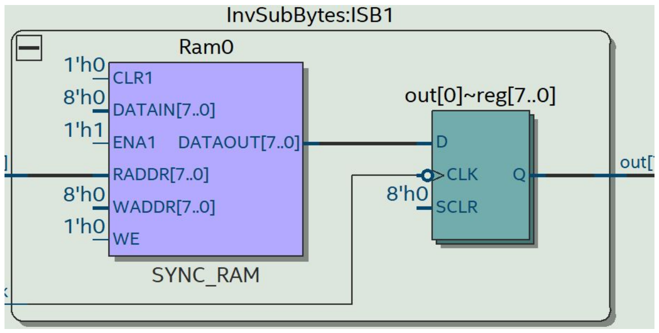
[7:0] in,

Outputs:

[7:0] out

Purpose & Description:

InvSubBytes would be synthesized into a RAM. In the always_ff logic, a mapping table of [7:0] in can return a result [7:0] out at the negedge of clk. 32 InvSubBytes work together to form the InvSubState.



Qsys Modules:

Module: nios2_gen2_0
Exports:
Description:

This is an economy version processor we allocate for NIOS II. It handles all the logical operations and computations our program might use. One more thing to notice is that this processor also supports JTAG Debug, which means we can debug using "printf" in Eclipse.

Module: onchip_memory2_0
Exports:
Description:

This is the onchip_memory module for our system, which is always used as cache or data buffer. On-chip memory is relatively "expensive" since it required a lot more logic elements, but it can significantly improve the memory operation speed compared on external memory chips.

Module: sdram
Exports: sdram_wire
Description:

This is the SDRAM controller for NIOS II CPU. SDRAM need to be refreshed every time as we do memory operations, we don't need to write a driver by ourselves, because this existing SDRAM IP module can be used in this case.

Module: sdram_pll
Exports: sdram_clk
Description:

Pll stands for "Phase Lock Loop." As suggested by the name, it's used to generate a second clock signal shifted by an amount. As explained in detail in the following subsection(INQ Question), this block is necessary to prevent skew operation.

Module: sysid_qsys_0
Exports:
Description:

This system ID checker is to ensure the compatibility between hardware and software. This module will give us a serial number, which the software loader checks against when we start the program. This prevents us from loading software onto an FPGA which has an incompatible NIOS II configuration.

Module: jtag_uart_0
Exports:
Description:

This module is used to let the FPGA board communicate with the PC. Thus the “scanf” and “printf” function in C program can be effectively implemented, and we can send data as well as read the result to/from the FPGA board.

Module: AES_Decryption_Core
Exports: aes_export
Description:

This module is our custom IP core based on the avalon_aes_interface.sv. Thus, the registers in the interface can be memory mapped to the NIOS II system. It provides us a more direct way to access and operate the registers values in the C program. It also enable us to create our own IP library.

Module: TIMER
Exports:
Description:

This module is used in the benchmark section. The interval timer can send periodic interrupts. So the system could calculate the computation speed based on that periodic interrupts.

Annotated Simulation of the AES decryptor

a. Show input encrypted message, the input plaintext, the output decrypted message and the current state of the state machine.

Post-Lab Questions

a. Fill out the design resources and statistics table

Design resources and statistics

LUT	5851
DSP	0
Memory (BRAM)	571392 bits
Flip-Flop	2869
Frequency	
Static Power	

Dynamic Power	
Total Power	

b. Answers to the post-lab questions

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 expect the hardware decryption could be way faster than the software encryption, because the NIOS II CPU which runs the software is designed for low-performance tasks (~0.15 MIPS/MHz), and pure hardware implementations of decryption units are able to process the entire 128 bits parallelly within just one clock cycle.

Function (Implementation Method)	Benchmark Result
Encryption (Software)	0.3628 KB/s
Decryption (Hardware)	83.3333 KB/s

As shown in our benchmark results, the hardware decryption is 230 times faster than the software encryption, and it validates our expectation.

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

Conclusion

Our AES encryptor and decryptor are able to work appropriately. In our system, the Plaintext can be correctly encrypted to its Ciphertext, and the Ciphertext can also be decrypted to the Plaintext, at the same time the first 16 bits and the last 16 bits of the key can be shown on the hex display. The software encryption speed is 0.3628 KB/s, and the hardware decryption can goes to 83.3333 KB/s. The result of this lab helps us to understand the advantage of speed in hardware implementation, and explore more potential application for FPGA (as acceleration hardware). We think the overall design of this lab is good, and it doesn't need to be modified in the future.