ECE385

Spring 2020

Experiment #9

SOC with Advanced Encryption Standard in SystemVerilog

Yuantao Lu, Yizhen Lu

Section ABD

TA: Nicholas Cebry, Wenjie Pan

Introduction

In this lab, we implement the AES encryption process using software and AES decryption using hardware. The process leads us through the mechanism beneath standard encryption/decryption and gives a direct comparison on how to implement the same thing using different approaches(software & hardware) and how they behave differently.

Written Description and Diagrams

Written description of software encryptor

The software part strictly followed the provided AES tutorial and implemented the AES encryption algorithm. There are several helper functions implemented for this algorithm: *KeyExpansion, AddRoundKey, SubBytes, ShiftRows and MixColumns.* The main function first lets the user input the 128-bit message and the key in ascii form and calls the encryption function. The encryption function first converts the ascii form message and key into normal char array form using the provided *charsToHex* helper function. Then the *KeyExpansion* helper function is called to generate a group of intermediate keys for the next step. Those intermediate keys are generally used for making *AddRoundKey* steps. Generally, we will go through the procedure ten times, adding the first key, so there are eleven intermediate keys generated, with each key having 128 bits. After the keys are generated, we go to the main loop part. In this part, we perform *AddRoundKey, SubBytes, ShiftRows and MixColumns* once in a loop. After looping 9 times, for the last time, we would not do *MixColumns* but add the corresponding round key at last. Now what is left to do is just making the result in the form of a group of 32 bit values.
The *KeyExpansion* function takes the original key as the first round key, and the following round keys are generated using the previous round keys. Each 128-bit word is made of four 32-bit words, each word w[i] = w[i-1] and w[i-4] in this case. For the word entries that are the multiple of four, they need to go through a cyclic shift and a SubWords function using the provided lookup table.
The *AddRoundKey* adds the current key value by the key in the corresponding round key generated by the *KeyExpansion.*

The *SubBytes* function performs a non-linear transformation on current state by taking the multiplicative inverse in Rijndael's finite field. The result can be found in the provided table.
The *ShiftRows* function performs a cyclic shift on current state.
The *MixColumns* function performs separate invertible linear transformations that the fours bytes in a word are linear combined to form a new word. There are shortcuts for this step and some of the results can be found in the provided table.

For the NIOS-II, all the C code that implements the encryption is running on NIOS-II. It uses the JTAG-UART module to create a terminal for the user to interact with the main function through scanf. There are also pointers in the C code that points to the port address of Avalon. Through the pointers, we would be able to control the corresponding registers in Avalon to communicate with the hardware. The necessary messages showing on the FPGA board are displayed by hardware but controlled by software that writes the messages into the corresponding registers.

Written description of hardware decryptor
The decryption in hardware followed the similar steps in encryption in software. First, it generates a series of round keys that is identical to the encryption round keys using the provided *KeyExpansion* module. The *InvSubBytes*, *InvShiftRows* and *InvMixColumns* modules are also given. Those modules perform the inverse operation of the *SubBytes, ShiftRows and MixColumns* functions in encryption. We did not create a seperate *AddRoundKey* module but write its logic in the combination logic in our state machine. The procedure for the decryption follows the encryption as well, except each step we perform those inverse functions in the reverse order compared to the encryption algorithm. We implemented a state machine that corresponds to each state of the decrypted message. The state transition happens after one of the operations is finished(each time the message was changed) As a result, there are about forty states representing all the transitions of the decrypted message.

The decryption module is called in the hardware/software interface and is controlled by the software. The process is controlled by the *Start* and *Done* signals. After the message and the key we wanted to decrypt are written into the corresponding Avalon registers, we set the start register so the hardware state machine starts to run. In the middle of the state machine, the *Done* signal is always zero, once the decryption is finished, the result will be written into the corresponding registers in the interface and the Done register will be set to 1. Before reading *Done* to be one, the software will wait in an infinite loop. After the loop, the software will read the result from the registers and return the decrypted message.

Written description of hardware/software interface

This is the module we designed to control the data flow from the hardware and the software. Unlike previous labs, instead of the tristate buffer and the PIO port, we used registers in Avalon to perform the data exchange. In the interface, the first thing to do is to create 16 32-bit registers. Based on the control signals, the input data will be written into corresponding registers and the output will be values in some corresponding registers. The hardware decryption module AES.sv is inside this module so this module can control the decryption logic directly(using the values of registers as input).

The first four registers are used for input keys; the second four registers are used for input encrypted messages; the third four registers are used for the result decrypted message; the second last register is for the input *Start* and the last register is used for output *Done*.

In platform designer, we designed the Avalon with all the input and output signals corresponding to the input and output signals of this module following the provided Introduction to the Avalon-MM interface. It has a port address, which can be read from the software. Since the Avalon has no cached memory, the registers can be directly accessed through the port address adding an offset. For instance, the 0th register's address is just port[0]. In this way, we can read from and write into the registers from the software conveniently.

Block diagram

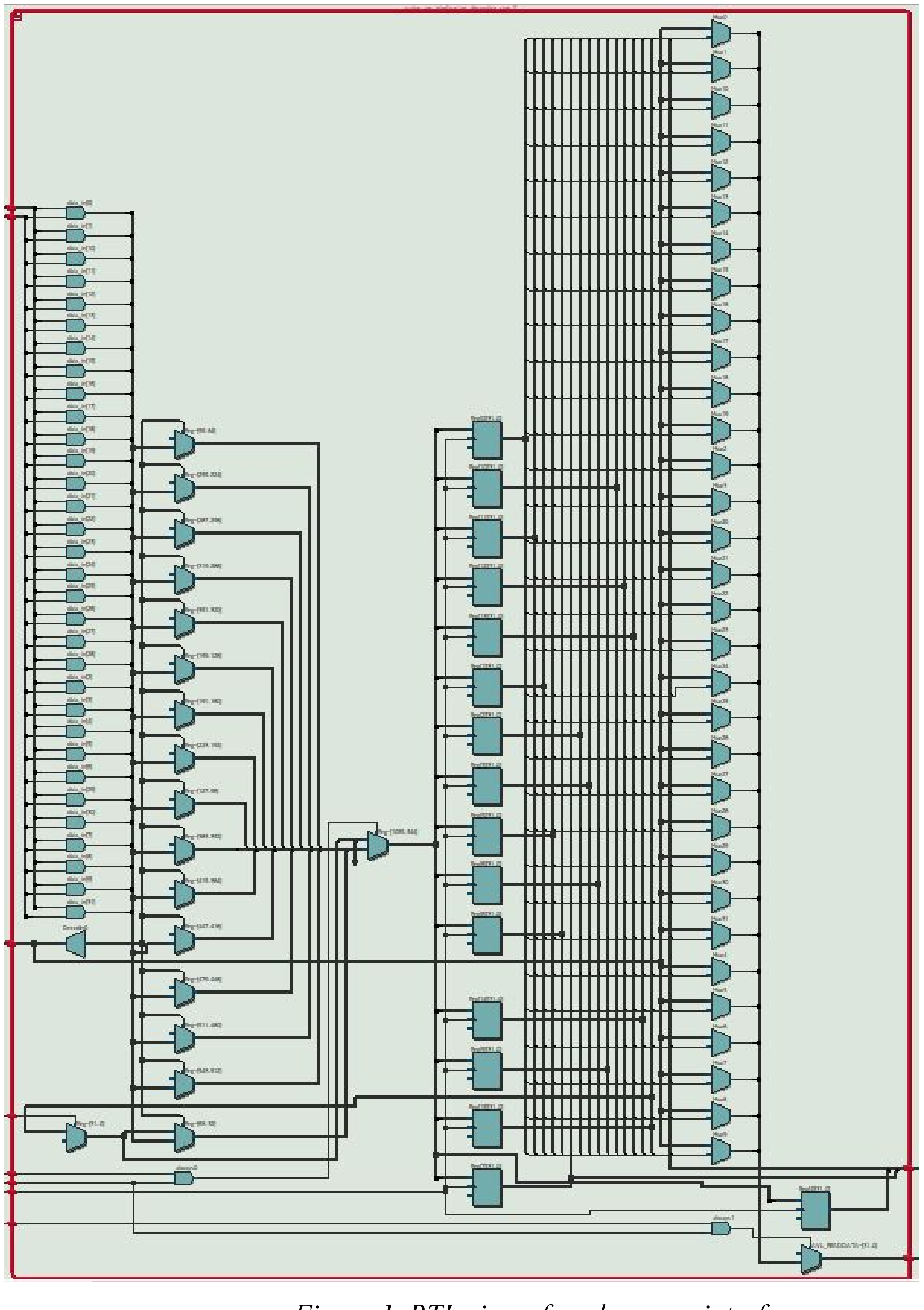


Figure 1. RTL view of avalon_aes_interface.sv

State Diagram of AES decryptor controller

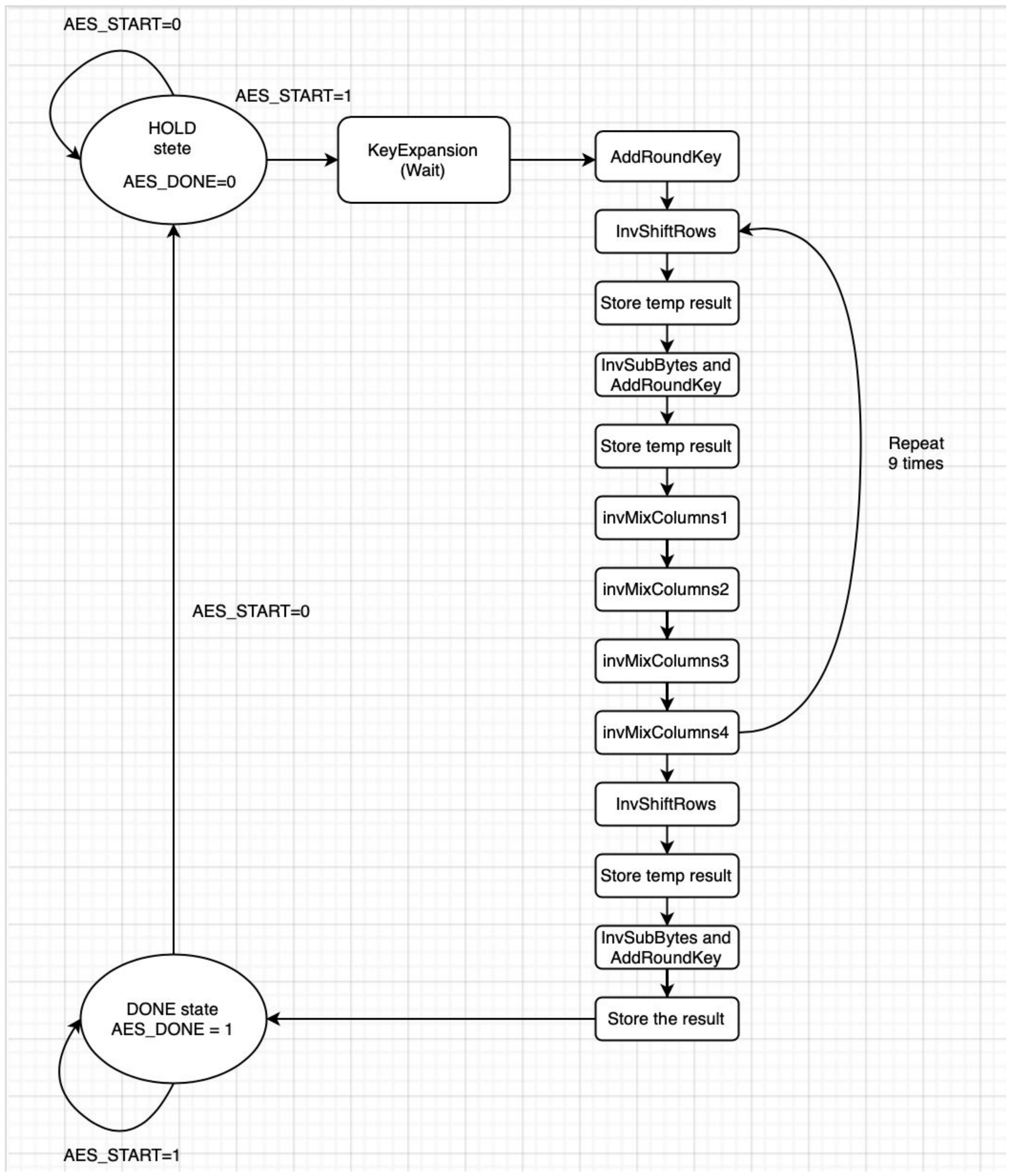


Figure 2. State Diagram for AES.sv

Module descriptions

Module: *lab9_top.sv*

Input: *CLOCK_50, [1:0] KEY, [31:0] DRAM_DQ.*

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, [31:0] DRAM_DQ, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK.*
Description: This module is the top level module for this lab. It generally represents the lab 9 system on chip and several HEX drivers that output the intended output message on the LEDs on the DE2 board.
Purpose: This module gives a top level view of the lab, it includes the lab9_soc module that is generated through the platform designer. It gives a clear view of the hardware of this lab and all the corresponding inputs and outputs.

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_WRITEDATA.*
Output: *[31:0] AVL_READDATA, [31:0] EXPORT_DATA.*
Description: This module represents the Avalon-MM interface that is described in the Introduction to Avalon-MM interface from the course website. It matches the block generated from the platform designer that is generated manually following the guide. All the inputs and the outputs are matched and the block has its own port address.
Purpose: This module is used to control the data flow between the software and the hardware part of this lab. It has sixteen 32-bit registers, each can be accessed by the software through the port address and the hardware directly. The data exchange happened by the registers. The *EXPORT_DATA* of this module will output the desired message to the LEDs on the DE2 board.

Module: *AES.sv*
Input: *CLK, RESET, AES_START, [127:0] AES_KEY, [127:0] AES_MSG_ENC.*
Output: *AES_DONE, [127:0] AES_MSG_DEC.*
Description: This module is used as a part of the *avalon_aes_interface.sv* to get the necessary outputs for it. It also includes all the sub modules related to the hardware decryption to perform the hardware decryption. Once the input *AES_START* is one,

the module will perform hardware decryption on *AES_KEY* and *AES_MSG_ENC*, the *AES_MSG_DEC* will be the final output.
Purpose: This module serves as the main module for hardware decryption. It is a state machine that processes the hardware decryption in many states. In each state, the state machine outputs necessary signals that control the submodules that work as the subfunctions for hardware decryption. It is also a part of a larger state machine that is controlled by the software. The main input and output for the larger state machine are *AES_START*, and *AES_DONE*.

Module: *hexdriver.sv*
Input: *[3:0] In*
Output: *[6:0] Out*
Description: This module transforms the hex data type to be the data type matching the LEDs.
Purpose: This module displays the *EXPORT_DATA* output from the module *avalon_aes_interface.sv* on the LEDs of the DE2 board.

Module: *SubBytes.sv*
Input: *[7:0] in.*
Output: *[7:0] out.*
Description: This module is used as a part of the AES module that performs the Decryption operation. When the stata machine outputs some signals, the module will be taking the input and outputs the result.
Purpose: This module performs the subBytes operation that is a part of the Decryption. It generally checks the input and outputs the value in the provided s_box array corresponding to the input.

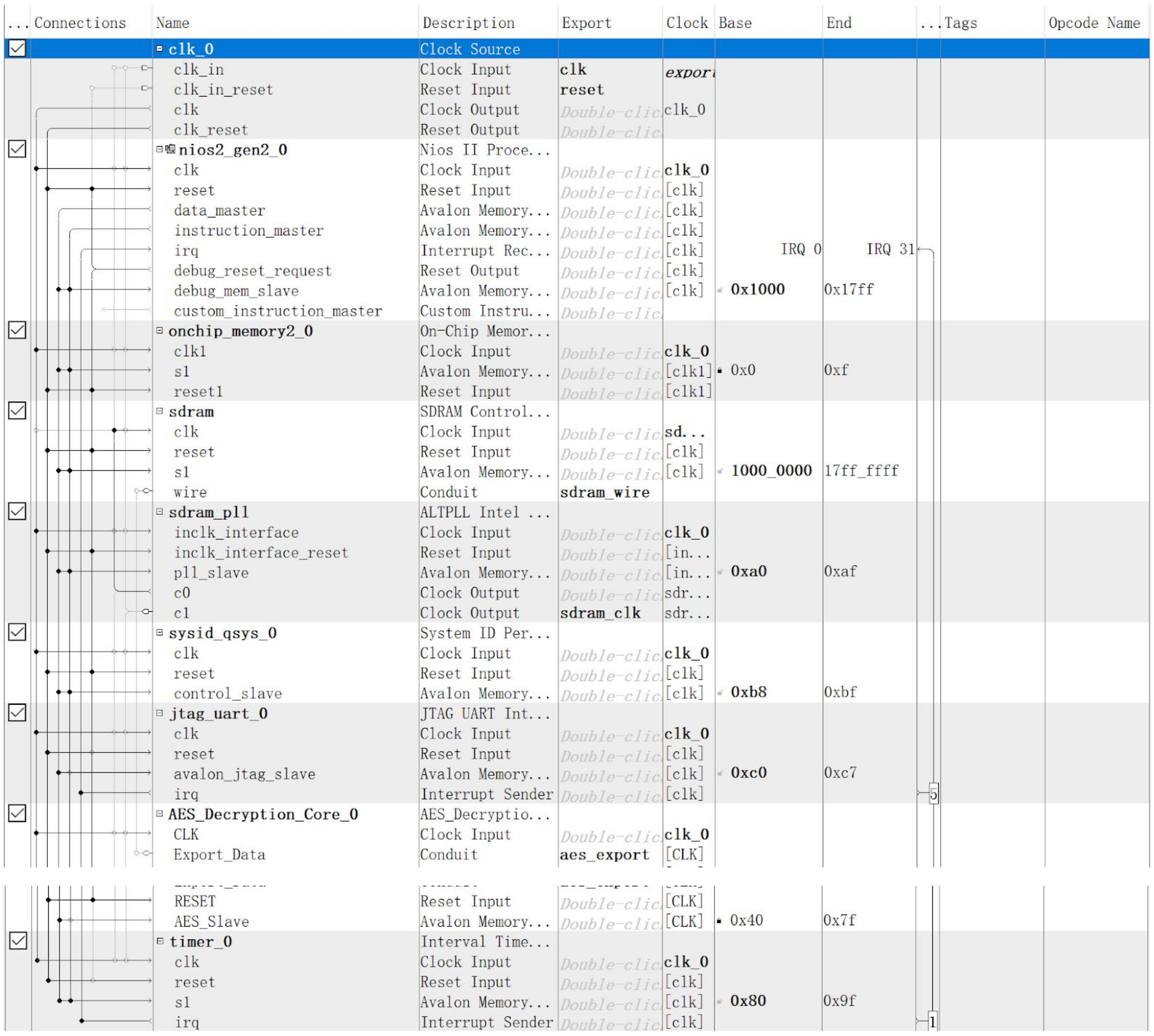
Module: *KeyExpansion.sv*
Input: *[7:0] clk, [127:0] CipherKey.*
Output: *[1407:0] KeySchedule.*
Description: This module is used as a part of the AES.sv that performs the necessary operations for hardware decryption. It is triggered at the start of the AES state machine. The states of “Wait” at the start of the state machine are all

corresponding to the KeyExpansion steps. Note that this process takes several clock cycles to finish.
Purpose: This module corresponds to the KeyExpansion function in software. It takes a 128-bit key and generates a series of round keys that will be used in the AddRoundKey part of the state machine at every round. Every round a different round key would be used, and each round keys are generated from previous round keys starting from the original key.

Module: *InvShiftRows.sv*
Input: *[127:0] data_in.*
Output: *[127:0] data_out.*
Description: This module is used as a part of the AES module that performs the Decryption operation. When the stata machine outputs some signals, the module will be taking the input and outputs the result.
Purpose: This module corresponds to the inverse shift rows in the decryption step. It shifts the 32-bit word in the input cyclically to generate the output.

Module: *InvMixColumns.sv*
Input: *[31:0] in*
Output: *[31:0] out*
Description:This module is used as a part of the AES module that performs the Decryption operation. When the stata machine outputs some signals, the module will be taking the input and outputs the result.
Purpose: This module corresponds to the inverse mix columns step in the decryption process. This module only performs the mix column once for a word. So to perform an inverse mix column for a 128-bit message, we need to use this module four times.

Qsys generated blocks



...	Connections	Name	Description	Export	Clock	Base	End	...	Tags	Opcode Name
☑		clk_0	Clock Source							
		clk_in	Clock Input	clk	export					
		clk_in_reset	Reset Input	reset						
		clk	Clock Output	Double-clic	clk_0					
		clk_reset	Reset Output	Double-clic						
☑		nios2_gen2_0	Nios II Proce...							
		clk	Clock Input	Double-clic	clk_0					
		reset	Reset Input	Double-clic	[clk]					
		data_master	Avalon Memory...	Double-clic	[clk]					
		instruction_master	Avalon Memory...	Double-clic	[clk]					
		irq	Interrupt Rec...	Double-clic	[clk]	IRQ 0	IRQ 31			
		debug_reset_request	Reset Output	Double-clic	[clk]					
		debug_mem_slave	Avalon Memory...	Double-clic	[clk]	0x1000	0x17ff			
		custom_instruction_master	Custom Instru...	Double-clic						
☑		onchip_memory2_0	On-Chip Memor...							
		clk1	Clock Input	Double-clic	clk_0					
		s1	Avalon Memory...	Double-clic	[clk1]	0x0	0xf			
		reset1	Reset Input	Double-clic	[clk1]					
☑		sdram	SDRAM Control...							
		clk	Clock Input	Double-clic	sd...					
		reset	Reset Input	Double-clic	[clk]					
		s1	Avalon Memory...	Double-clic	[clk]	1000_0000	17ff_ffff			
		wire	Conduit	sdram_wire						
☑		sdram_pll	ALTPLL Intel ...							
		inclk_interface	Clock Input	Double-clic	clk_0					
		inclk_interface_reset	Reset Input	Double-clic	[in...					
		pll_slave	Avalon Memory...	Double-clic	[in...	0xa0	0xaf			
		c0	Clock Output	Double-clic	sdr...					
		c1	Clock Output	sdram_clk	sdr...					
☑		sysid_qsys_0	System ID Per...							
		clk	Clock Input	Double-clic	clk_0					
		reset	Reset Input	Double-clic	[clk]					
		control_slave	Avalon Memory...	Double-clic	[clk]	0xb8	0xbf			
☑		jtag_uart_0	JTAG UART Int...							
		clk	Clock Input	Double-clic	clk_0					
		reset	Reset Input	Double-clic	[clk]					
		avalon_jtag_slave	Avalon Memory...	Double-clic	[clk]	0xc0	0xc7			
		irq	Interrupt Sender	Double-clic	[clk]			5		
☑		AES_Decryption_Core_0	AES_Decryptio...							
		CLK	Clock Input	Double-clic	clk_0					
		Export_Data	Conduit	aes_export	[CLK]					
		RESET	Reset Input	Double-clic	[CLK]					
		AES_Slave	Avalon Memory...	Double-clic	[CLK]	0x40	0x7f			
☑		timer_0	Interval Time...							
		clk	Clock Input	Double-clic	clk_0					
		reset	Reset Input	Double-clic	[clk]					
		s1	Avalon Memory...	Double-clic	[clk]	0x80	0x9f			
		irq	Interrupt Sender	Double-clic	[clk]			1		

Figure 3. The Qsys generated blocks

clk_0:

This block generates the main clock signal for all other modules.

nois2_gen2_0:

This block represents the NIOS-II economic version. It is the microprocessor we used in this lab, which performs the computing and processing job in the hardware.

onchip_memory2_0:

It is the memory corresponding to the NIOS-II for loading and storing data that requires high-level performance.

sdram:

This block represents the sdram module. It makes a connection with the processor, so the system can use the memory specified by the sdram.

sdram_pll:

This block represents the PLL for the system. It controls a clock with a phase shift of 3ns prior to the main clock. In this way, the sdram will start processing before the clock edge, and the date would be faster processed after the main clock edge.

sysid_qsys_0:

This block represents the system id checker. The purpose of this block is to check the configuration of the hardware part and the software part to make sure the two matches.

jtag_uart_0:

This block creates a terminal as a user interface so users can interact with the software code. The messages can be printed into the terminal through printf and user input can be read into the terminal through scanf.

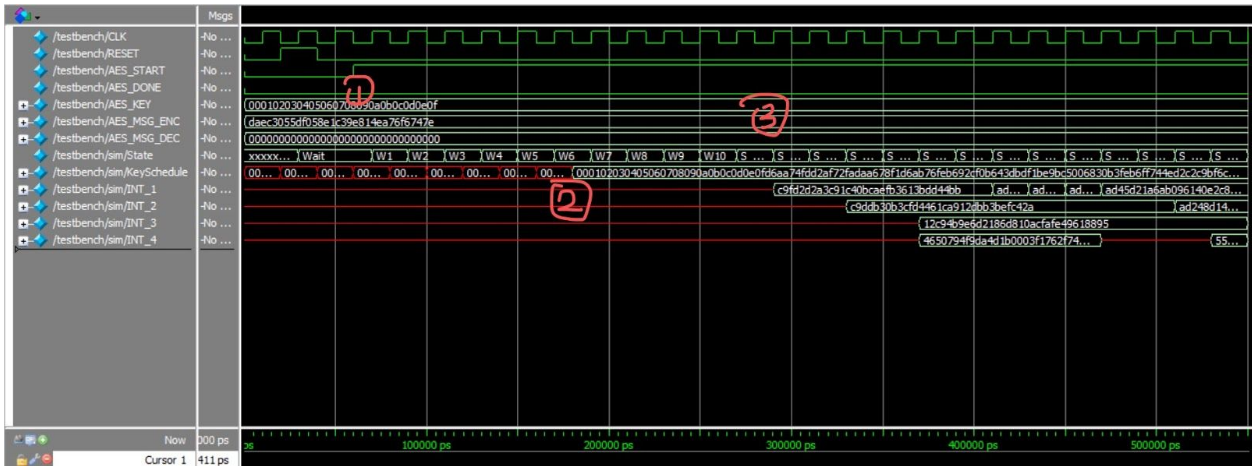
AES_Decryption_Core_0:

This block represents the *avalon_aes_interface* module that is used to control the hardware-software data flow. It is a manually generated block without using the block models. We can generate similar blocks in the same way to perform desired tasks. This hardware has a port, which can be accessed as a pointer in software. The data can be written into the address related to this pointer to realize the data exchange between software and the hardware.

timer_0:

This block is used to count the time between a starting and a stopping point in order to compute the speed of processing of software and hardware. The corresponding function is called in the main function in C and this block is the hardware corresponding to it.

Annotated Simulation of the AES decryptor



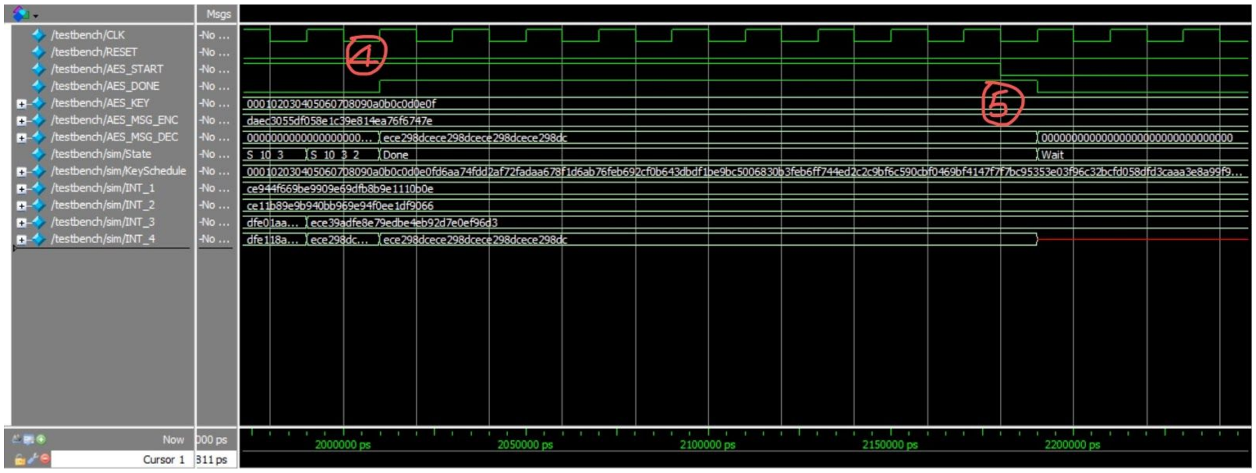


Figure 4,5. Sections of AES.sv Simulation Waveform

Markings:

①. *AES_START* is toggled to high, meaning that the module receives the signal to start the decryption process.

②. After a few cycles, the *KeyExpansion* module instantiated in *AES.sv* finishes generating the entire key schedule based on the given *AES_KEY*.

③. After the key schedule is ready, the state machine proceeds to begin the 10 rounds of decryption cycles.

④. The decryption process is complete. *AES_DONE* is high and the message text is sent out through the output *AES_DEC.*
⑤. Toggle *AES_START* to low resets the whole state machine, making it ready for a new decryption attempt.

Post-Lab Questions

1)

LUT	6047
DSP	N/A
Memory(BRAM)	571392 bits
Flip-Flop	3381
Frequency	54.67MHz
Static Power	68.12mW
Dynamic Power	76.67mW
I/O Thermal Power	102.54mW
Total Power	247.33mW

2)

- Which would you expect to be faster to complete encryption/decryption, the software or hardware? Is this what your results show?
  - Hardware is expected to be faster and the results agree with the expectation. Software runs at around 0.43 KB/s while hardware runs at 200 KB/s.
- If you wanted to speed up the hardware, what would you do? (Note: restrictions of this lab do not apply to answer this question)

- In this lab, the *InvMixColumns()* module is only allowed to be instantiated once. This means each InvMixColumns step in the decryption process will need to take at least four clock cycles to finish. Therefore, to enhance the performance, we can instantiate four such modules so that the InvMixColumns step will only need to take one cycle.
- If we don't consider saving energy and resources, we can instantiate modules everytime we need them, so that we won't take extra cycles to load the intermediate values in registers each time. In this way it would be the most time efficient.

Conclusion

The functionality of this lab is fine. The hardware performs the decryption process correctly and quickly even though we think our decryption logic was not the most efficient one. We noticed although the software can perform many various tasks, the speed for the software is usually slow to go through the cycles for the computers many times. Using hardware, although hard to write and can only perform limited processes, the speed is way faster than the software.

For the ambiguous part of this lab, we think the hardware design part for this lab from the platform designer is very problematic. We followed the procedure step by step but ended up with the bugs from the version of Quartus. We started with the 18.0 version of the Quartus but the generated module does not work. In the documentation, the bug was pointed out at the very end, and the format of the code for fixing the bug is wrong. So we have to do the design again using 18.1 because we didn't know the correct format at first. The generated HDL using 18.1 is also problematic, we have to change the names for the generated module multiple times to fix it. We think it is useful to put this bug at the beginning of the document instead of putting it at the end.

In addition, we think this lab is not that useful and we did a lot of unnecessary work. For instance, the encryption and decryption algorithm is not related to the hardware design from any perspective. The algorithm is good but the intermediate steps are not very useful. In most cases, we were just looking for data in the table provided. We also do not get the point to implement the encryption functions ourselves in C. It is very time consuming but we did not learn too much related to ECE385.

The core point for this lab is to let us write the same functions using hardware and software and comparing their speed to show how fast hardwares can be. It also taught us how to use Avalon to control the data flow between the software and the hardware, and manually generate the blocks from Qsys. There are many other useful and easier functions we could use to reach the goal of this lab. We could just write easy image processing functions in C and in hardware. This will not only show the hardware priority in speed, but also link our knowledge to VGA, which is very useful to the final project and not that time consuming.

We hope that lab9 of ECE385 can be changed to a more useful one in the future.