ECE 385

Fall 2016

Experiment #9

SOC with Advanced Encryption Standard in System Verilog

Neil Patel & Siddhant Jain

Lab AB1

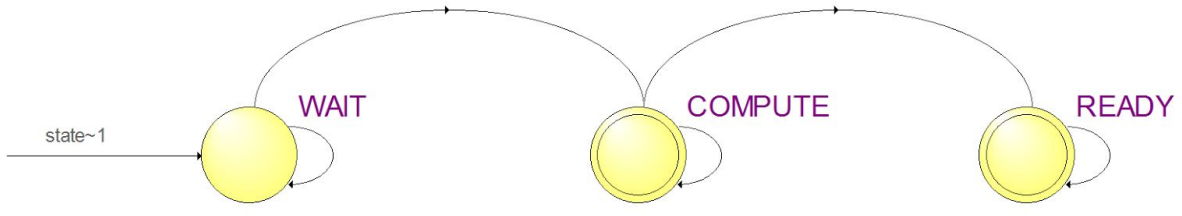
Conor Gardner

Introduction

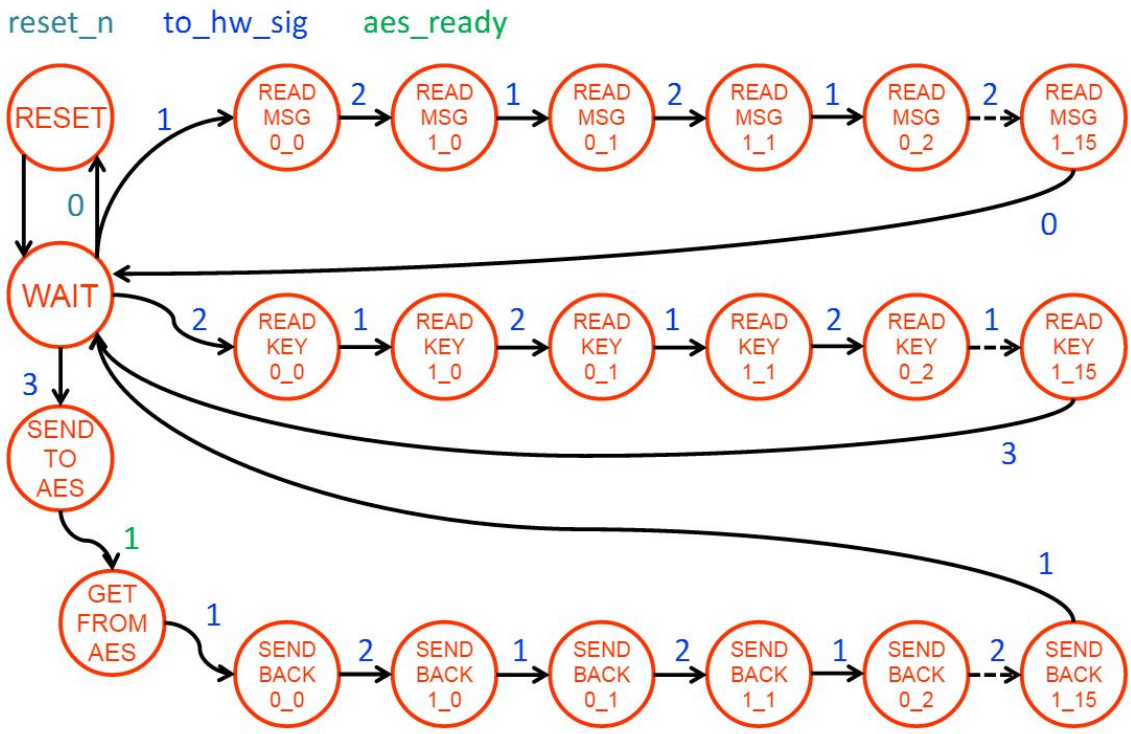
The purpose of Experiment 9 was to write SystemVerilog and C code to build 128-b Advanced Encryption Standard as an Intellectual Property core. In Week 1, we were tasked with designing a 128-bit AES encryption on the IP core. The second week's goal was to build the 128-bit AES decryptor using System Verilog and C. This lab was the first time we interfaced with an IP Core. An IP core is a block of reusable logic,code, or hardware design that is the intellectual property of a corporation or other entity. The way it is used is equivalent to a software library in C, which means that it can re-used many times in different applications. They don't serve as a full program but rather they target a specific use case that occurs many times. Ultimately, we learned how to build an IP core and how to incorporate with C and with the NIOS II processor. Putting all of this together helps us create a SoC. The end goal of this lab was to create a circuit such that it can decrypt user inputted data with a specific key to recreate the original intended message.

Written Description and Diagrams of the AES encryptor/decryptor

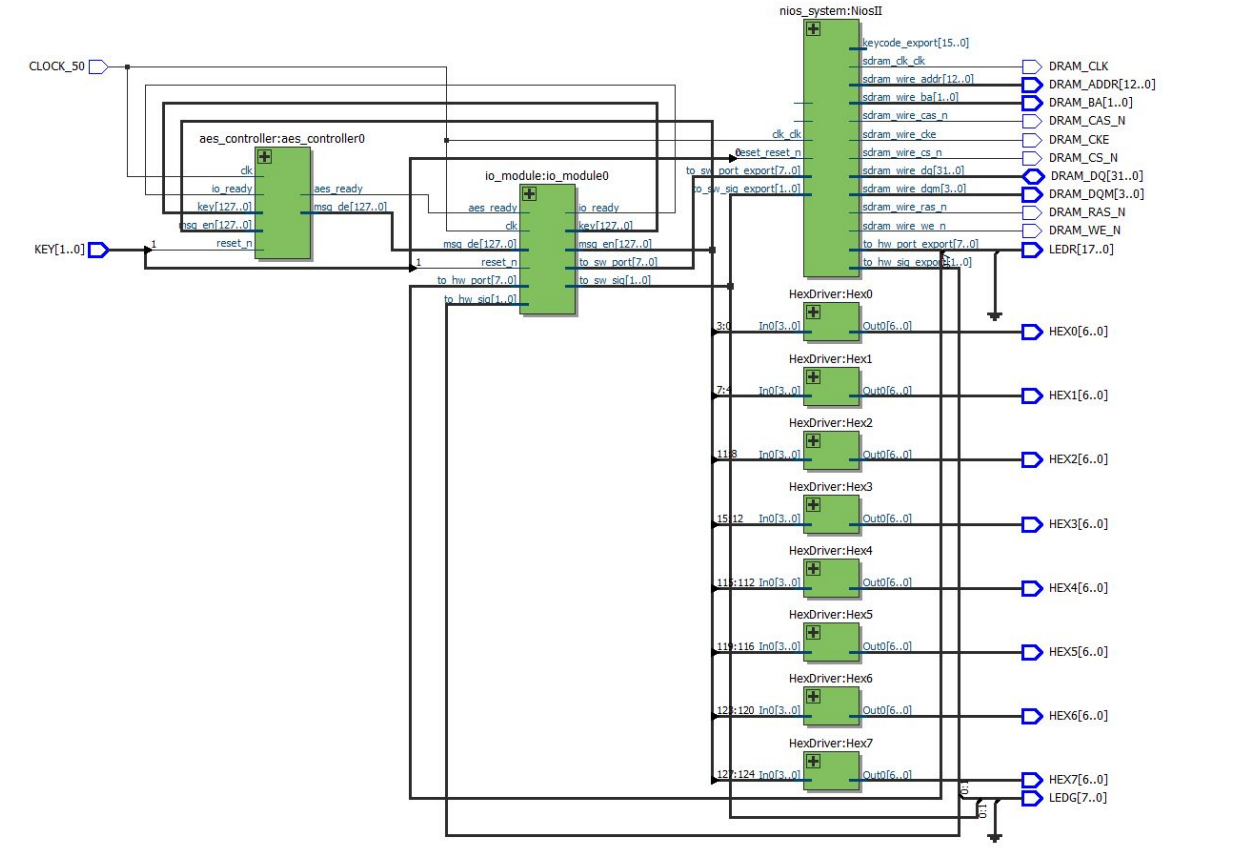
AES Controller



IO_State Diagram



Block Diagram



Software Encryptor Description

Our encryption process starts off in hello_world.c . Software data and signal lines show the when and what to encrypt. The main() function starts off by prompting for the reset button; any nonsense or leftover data is force cleared. The code loops; the first part of the loop is where the console asks the user for a message composed of 32 characters. Said message is then sent to the FPGA in the form of 16 1-byte pieces using the provided char-to-hex function provided by the ECE 385 course staff.The next printed message asks for an encryption key (the same key is

needed for decryption) Char-to-hex is used to transmit the key to FPGA. NIOS II expands the key using a KeyExpansion function; this turns the single key provided by the user into 11 keys for use during encryption. Now is when we actually start the encryption process! AddRoundKey is executed once; this function XOR's the message and the first of 11 keys. Now, for the next 9 generated keys (After being run through AddRoundKey), the message with undergo subBytes,ShiftRows,Mixcolumns, and AddRoundKey. SubBytes takes the message bytes and substitutes then in for the matching Rijndael byte (which is a provided matrix). ShiftRows shifts rows in increments by row number. MixColumns uses gf_mul[][] (a given function) to call upon pre-calculated matrix multiplication results, which leads to a much faster program. AddRoundKey XOR's said output and the next AddRoundkey key. The final time, the functions except for AddRoundKey will be executed again, with the result being printed to the console.

Hardware Decryptor Description

Although we didn't write the hardware decryptor, the way its code functions is literally the exact same as the software encryptor but reversed! We would have to write a state controller that handles the order and number of runnings of various modules. Our C function/ hardware passes in the encrypted message. The key is then run through a given module called inverseKeyExpansion; it generates 11 keys, just like the KeyExpansion in the software part, but with an inverse algorithm. The next step is for the code to be run through InvRoundKey, inverseShiftRows, and inverseSubBytes, which, as the name states, are the inverses of the functions that we wrote in the software. Just like in the software, these functions would be run 9 times and do the inverse of the functions that are in the software.

IO Module Written Description

IOmodule.sv has certain inputs and outputs that help it determine how and what to communicate between the software and the hardware. The IOmodule.sv also serves as a sort of "handshaking" between the hardware and software; the send_back and got_ack states wait for each other to happen in pieces and make sure the others do happen. Essentially, the software won't put new messages onto the hardware port until it receives acknowledgement that the hardware received its previous part of the message; this is necessary b/c the port is 8 bits but the message itself is 128 bits. In conjunction, hardware won't move to the read state until it receives confirmation from the software that a new message has been sent!

Modules

Module: aes_controller.sv

Inputs: clk, reset_n, [127:0] msg_en, [127:0] key, io_ready

Outputs: [127:0] msg_de, aes_ready

Description: Although we didn't really use it because we did not do part 2 of the lab, aes_controller implements a much simpler (the AES) fsm that, when io_ready is 1, takes in the encrypted message and uses the AES module to decrypt it.

Module: io_module.sv

Inputs: clk, reset_n, [1:0] to_hw_sig, [7:0] to_hw_port, [127:0] msg_de, aes_ready

Outputs: [1:0] to_sw_sig, [7:0] to_sw_port, [127:0] msg_en, [127:0] key, io_ready

Description: IO module is described above.

Module: AES.sv

Inputs: clk, [0:31] Cipherkey_0, [0:31] Cipherkey_1, [0:31] Cipherkey_2, [0:31] Cipherkey_3

Outputs: [0:31] keyschedule_out_0, [0:31] keyschedule_out_1, [0:31] keyschedule_out_2, [0:31] keyschedule_out_3

Description: Because we only did lab part 1, our AES.sv is basically empty minus what was given to us already. In reality, the AES.sv file is a complicated FSM that is supposed to handle the task of decryption; it takes in the key and encrypted text; and when Run comes in as 1, it begins the decryption. Some of its states use other modules to finish whatever needs to be done during that specific state.

Module: KeyExpansion.sv

Inputs: clk, [0:127] Cipherkey

Outputs: [0:1407] KeySchedule

Description: The KeyExpansion.sv file was given to us by the Course Staff as it is complex to implement key expansion. The purpose of KeyExpansion.sv is to do the inverse of the KeyExpansion function in software.

Module: InvMixColumns.sv

Inputs: [0:31] in

Outputs: [0:31] out

Description: The InvMixColumns.sv file was given to use by the Course Staff as it is complex to implement Rijndael's algorithm. The purpose of InvMixColumns.sv is to do the inverse of the MixColumns function.

Module: lab9.sv

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

Inouts: [31:0] DRAM_DQ

Description: The Lab9.sv is the top level file that connects the NIOS II Processor to all the other drivers and blocks.

Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: HexDriver.sv converts 4-bit input into 7-bit hex code that is displayable. It also defines how the FPGA display hex inputs.

Module: nios_system.v

Inputs: clk_clk, reset_reset_n, [7:0] to_sw_port_wire_export, [1:0] to_sw_sig_wire_export

Outputs: [1:0] sdram_wire_ba, sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n, [7:0] led_wire_export, sdram_clk_clk, [12:0] sdram_wire_addr, [3:0] sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n, [7:0] to_hw_port_wire_export, [1:0] to_hw_sig_wire_export

Inouts: [31:0] sdram_wire_dq

Description: This is the NIOS II module. The purpose of this module is to take the inputs of the PIO modules(to_sw_sig and to_sw_port) and designate the outputs through hardware (to_hw_sig and to_hw_port). This also enable console activity through the jtag_uart module. .

Annotated Simulation of the AES decryptor

We were unable to complete the decryptor (Part II) so we do not have the simulation to present.

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 decryption to be faster because it utilizes the hardware. We can't actually test it as we didn't get the decrpytor to work. However we were curious so researched to see what the outcome is. We learned from *Info-Security* that "It also requires minimum configuration and user interaction and does not cause performance degradation." (Brecht). From this we

understand that hardware is quicker as it does need software interaction and the performance is optimal versus Software encryption is easier to implement how it is slower.

For encryption, we were able to do 10,000 encryptions in approximately 33 seconds. At 64 bits per an iteration (32 for the key, and 32 for the message), this calculates out to a rate of 2.42e-3 MB/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)

The easiest way to speed up our hardware given that we do not have to deal with the restrictions of the lab has to do with the InvMixedColumns module. We were restricted by the instructions to only have 1; if we instantiated more instances of these modules, we could do our calculations faster because InvMixedColumns works on data inputted from other sources, not the output of the previous InvMixedColumns.

Design and Resource Table

LUT	3381
DSP	N/A
Memory (BRAM)	93312
Flip-Flop	2393
Frequency	90.05 MHz

Static Power	102.07 mW
Dynamic Power	0.95 mW
Total Power	178.40 mW

Conclusion

Our project worked for part 1. We were able to make the software encryptor. ALthough we weren't able to complete part 2 on the decryptor we understand how it is suppose to work. Once we got the encryptor to work, our mind was blown because it felt like something a top secret NSA agent would make but it was something we were able to do it in college. The difficult part to it was understanding the state diagram and what each function was suppose to do. Ultimately completing lab 9 part 1, the encryptor was really cool and enjoyable as we took something that is standard practice and built it in class.

Citations

Info-Security, Daniel Brecht,

http://www.infosecurity-magazine.com/magazine-features/tales-crypt-hardware-software