Lab 9 Report

ECE 385

Friday (ABB)

11:00AM-1:50PM

SOC with Advanced Encryption Standard (AES) in SystemVerilog

Ritvik Avancha and Steven Phan

(rra2 and sphan5)

Table of Contents

Introduction

As the final official lab in the class, with the exception of the final project, we implemented a 128-bit Advanced Encryption Standard (AES) using SystemVerilog as an IP core. During week 1, we explored the software side of AES encryption by writing all the necessary functions for the AES encryption process in C. The only "hardware" aspect of this is communicating with the hex displays in order to display the key typed into the eclipse console. Week 2 we wrote the AES decryption process completely in SystemVerilog with the only "software" aspect being to retrieve the encrypted message/key from week 1, storing it in a memory, and having a decryption driver function in the main.c file which interacted directly with hardware signals (i.e. AES_START/DONE) in order to start the hardware decryption process. Writing the process in HW vs. SW allows us to compare the respective performances.

Written Descriptions and Diagrams of AES Encryptor/Decryptor

Software Encryption

The role of the NIOS 2 Processor (embedded processor) is to allow us to abstract away the process of AES encryption into a higher level language, like C. Essentially, using NIOS as the main GUI allows for keys and messages to be input via the eclipse terminal and encrypted with various functions written in main.c. The main function polls for keyboard input for both messages and keys, and then calls encrypt which is a driver function that calls various helper functions like MixCols, ShiftRows, SubBytes, AddRoundkey, KeyExpansion, and so on in order to run through the AES encryption algorithm and produce a encrypted message. For week 1, this message didn't have to be stored in the Avalon-MM interface, but since the key had to be displayed via hex displays, we decided to store the encrypted message in the register file alongside with the typed in key. When benchmarking this process, it appears to be really slow in comparison to week 2's speed which performed roughly 400x better.

Hardware Decryption

The second week of the lab was to undo the software encryption week 1 but in hardware. Essentially, we had to implement a hardware decryptor. In order to do this, we developed a FSM

that considered the stated lab restraints (KeyExpansion and InvMixCols can only be instantiated once). The way the FSM works is that it starts in a WAIT state and as long as AES_START = 0, it stays there. AES_START is set to 1 only when the software is done encrypting a message, and this is reflected in the decrypt function within main.c. Once AES_START gets set to 1, it enters the decryption states. In encrypt, we iterated through the key expansion starting from the original cipher key to key10, but in decrypt we had to go through it backwards. In addition to that, KeyExpansion.sv takes multiple clock cycles to finish, so we accounted for that by first entering the key expansion (KE) state, and waiting a few cycles for it to continue. The next state is loading the encrypted message into a temporary state register. Once this was done, the next state would be to add key10 to the state, then perform InvShiftRows (round++) → InvSubBytes → AddRoundKey (grabs a portion of expanded key schedule based on round) → InvMixCols (x9). In order to adhere to the restriction of only instantiating one instance of InvMixCols, we had 5 states dedicated to InvMixCols that would select a current column of the state register, perform an operation on it, and output it as an inverted column. This goes into a temporary register that holds the overall output of the 4 inverted columns and is loaded into the state on the 5th state (4 for each column, last state is to load it into the state register). Once the round count hits 10, AES_DONE gets set to 1 and the software reads the decrypted message and sets AES_START = 0. This puts the FSM back into the WAIT state for the next message to be typed in.

Hardware/Software Interface

The register file is 16x32 bit, where 2 registers are unused. Since this version of the AES is 128 bits, we had to split the keys, encrypted message, and decrypted message into 4 registers each. Registers[3:0] hold the typed in keys, [7:4] hold the encrypted message done in week 1, and [11:8] hold the final decrypted message done in week 2. The way data is sent between NIOS and the hardware decryptor is as follows: software computes encrypted message (converted to hex of course) → sends this to the register file in the avalon_aes_interface through the Avalon-MM slave port → software says AES_START = 1 which is written into the register file (Register[14]) via the Avalon-MM → triggers the decrypt FSM to start → decrypt message finishes → stores result in Registers[11:8] → AES_DONE is set to 1 is written into

memory-mapped Register[15] which triggers software to read the message → then software AES_START = 0 puts the FSM back to its WAIT state for the next cycle. Essentially, all signals that trigger the interaction between SW/HW act in a “handshake” fashion. One side waits for the other to send a signal through the Avalon-MM slave port in order to do the next step. This can be summarized by Figure 2.1.

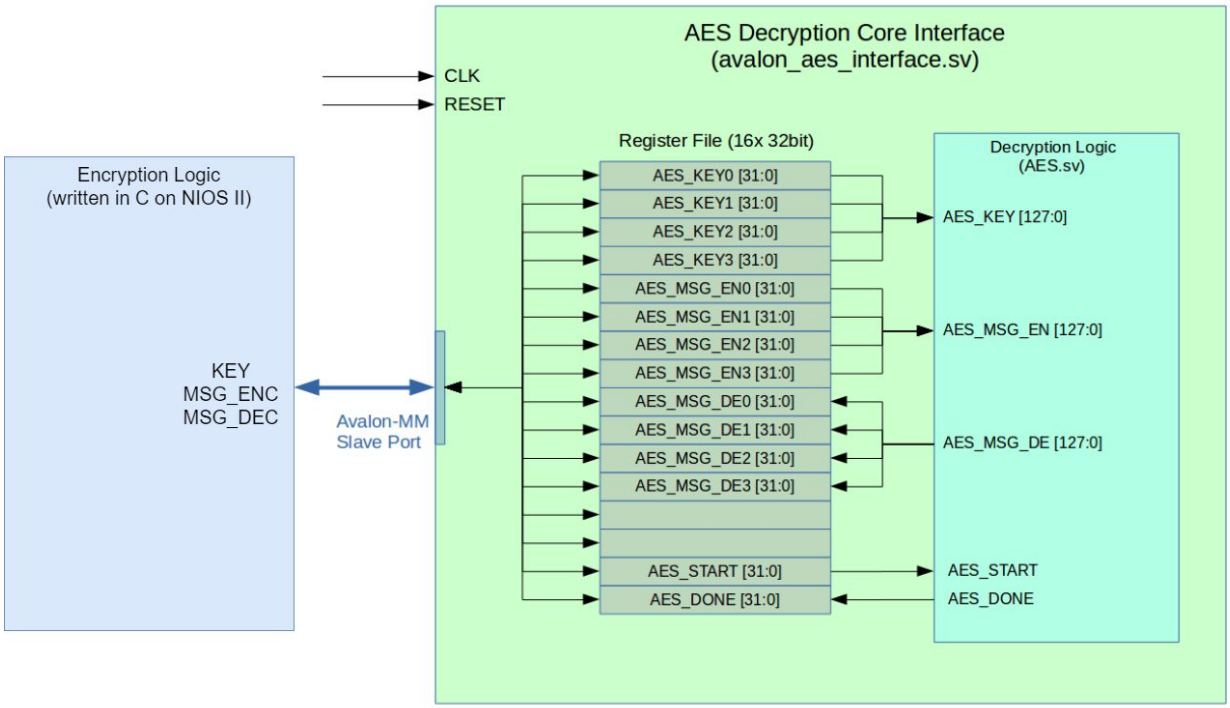

Figure 2.1: HW/SW Interface

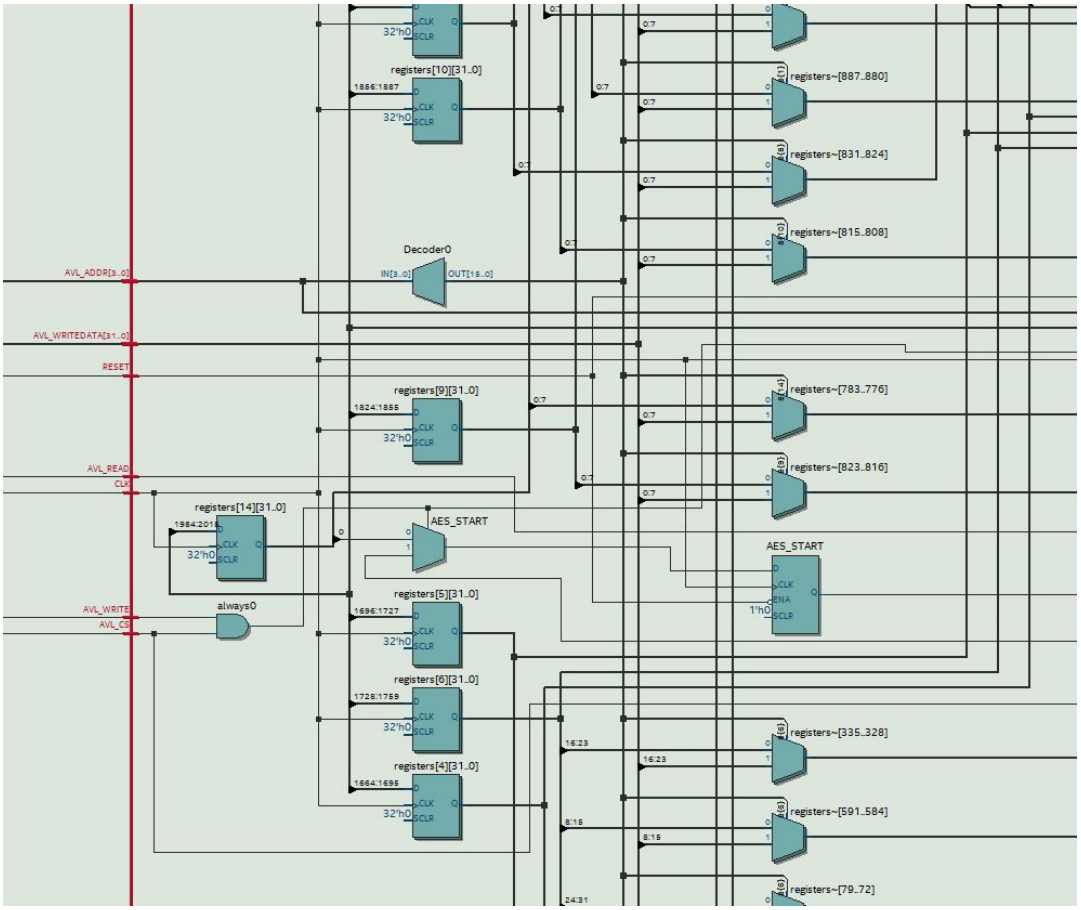

Figure 2.2: Avalaon_Aes_Interface Block Diagram (Part I)

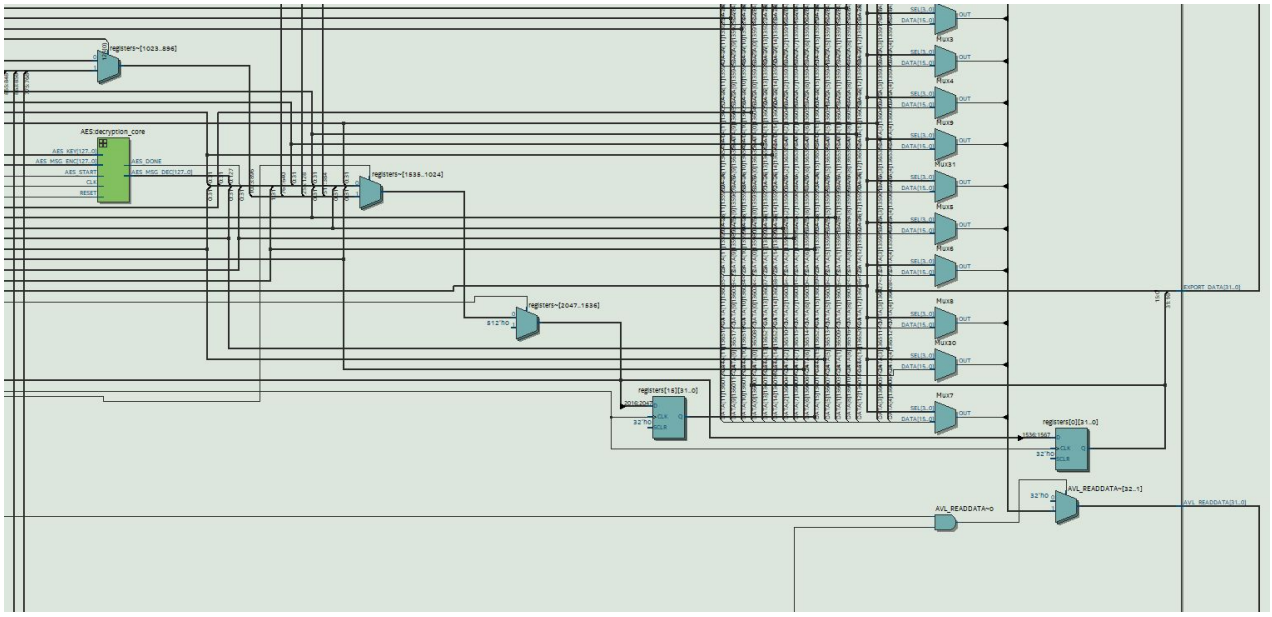

Figure 2.3: Avalon_Aes_Interface Block Diagram (Part II)

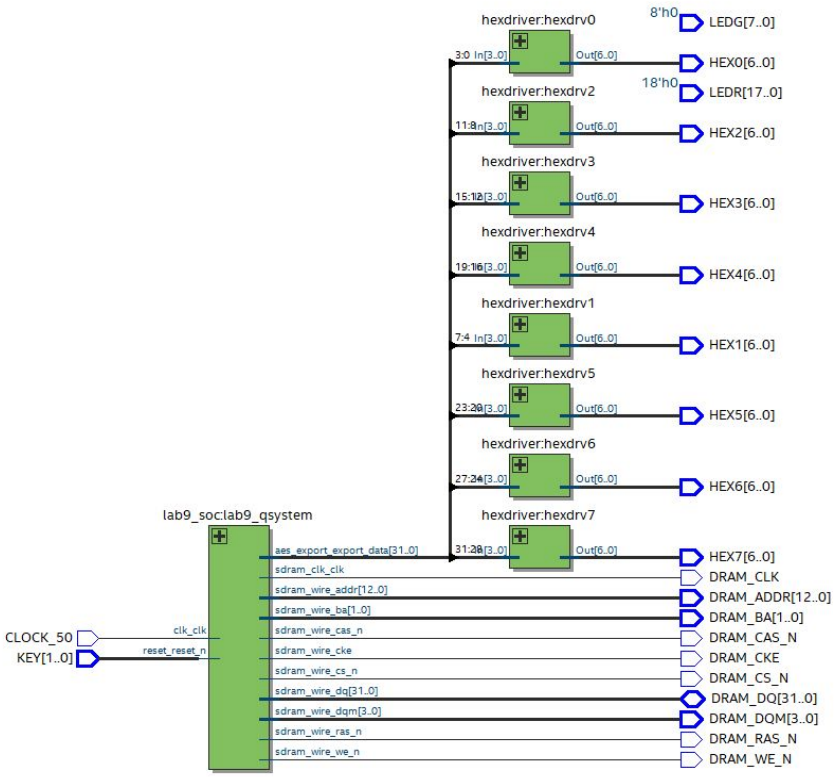

Figure 2.4: Top Level Block Diagram

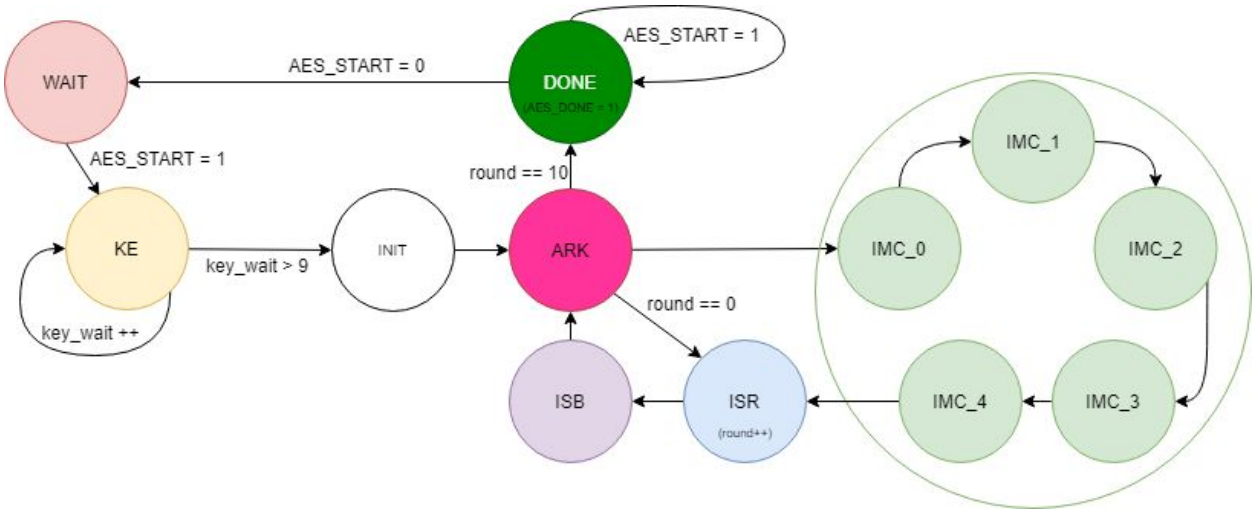

Figure 2.5: State Diagram

Descriptions

Module Descriptions

Module: **lab9_top.sv**
Input: CLOCK_50, [1:0] KEY
Inout: [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, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK
Description: Top level entity for the whole lab
Purpose: Top level instance for the lab used for instantiating the lab9_soc module and eight instances of the hexdriver module.

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: Instantiates the AES module and 16 registers as seen in Figure 2.1
Purpose: Interface the avalon module with the AES module with intermediary registers to hold the encrypted and decrypted values along with the key

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: Instantiates all the necessary function and control modules for decryption, such as: addRoundKey, InvSubBytes, InvShiftRows, InvMixColumns, KeyExpansion, and DEC_SYS
Purpose: Handles all the decryption logic before handing back control to the NIOS II system

Module: **addRoundKey.sv**
Input:[127:0] state, [127:0] roundKey
Output: [127:0] out
Description: Takes in the current key from the msg_dec register along with the current round key and outputs a bitwise XOR for the inputs
Purpose: The add round key state is used in order to undo the bitwise XOR operation during encryption

Module: **DEC_SYS.sv**
Input: CLK, RESET, AES_START
Output: AES_DONE, [3:0] round, [1:0] LD_FUNC, [1:0] CURR_COL, LD_IMC_REG, LD_MSG_DEC, INIT_REG
Description: Handles the control logic for the AES module
Purpose: Implements the state diagram illustrated in Figure 2.5 into the hardware

Module: **InvSubBytes.sv**
Input: clk, [7:0] in
Output: [7:0] out
Description: Using a given inverse substitution table, the module takes a two word input (8 bit value) and outputs a corresponding two word value.
Purpose: Undo the substitution operation during encryption

Module: **InvShiftRows.sv**
Input: [127:0] data_in
Output: [127:0] data_out
Description: Performs an inverse shift rows and outputs a 128 bit column major matrix.
Purpose: Undo the shift done during encryption process

Module: **InvMixColumns.sv**
Input: [31:0] in
Output: [31:0] out
Description: Performs the AES inverse mix columns on a column of the state of the decrypting message
Purpose: Undo the mix column done during the encryption process, column wise on a state.

Module: **KeyExpansion.sv**
Input: clk, [127:0] Cipherkey
Output: [1407:0] KeySchedule
Description: Takes in a cipher key (original key) and generates a key schedule (11 sets of keys)
Purpose: Generates {CipherKey, Key0-9, Key10} for Add Round Key at every round in the AES process.

Module: **hexdriver.sv**
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: Converts a 4-bit input into needed signals for 7-segment hex display on the FPGA board.
Purpose: Used to display the current values in PC/Switches/Memory contents in various states

QSYS Block Descriptions

clk_0: A 50 MHz clock used to clock all the synchronous components of the design
nios2_gen2_0: NIOS 2 processor runs the compiled C code stored in SDRAM
onchip_memory2_0: NIOS 2's 16 byte memory
sdram: Memory space used to store instructions and general storage
sdram_pll: Generates a 2nd clock that is out of phase with the main clock in order to ensure that when SDRAM refreshes, the input ports have time to stabilize
sysid_qsys_0: verification module for interfacing code with the board. It ensures that the code that is trying to get on the is meant for that board

jtag_uart_0: Allows for NIOS 2 to directly communicate with the host PC via USB Blaster (used generally for debugging purposes like printf statements and so on)
AES: Interfaces the software running on the NIOS II system with the hardware to allow for hardware decryption
TIMER: Allows for benchmarking of the software encryption and the hardware decryption

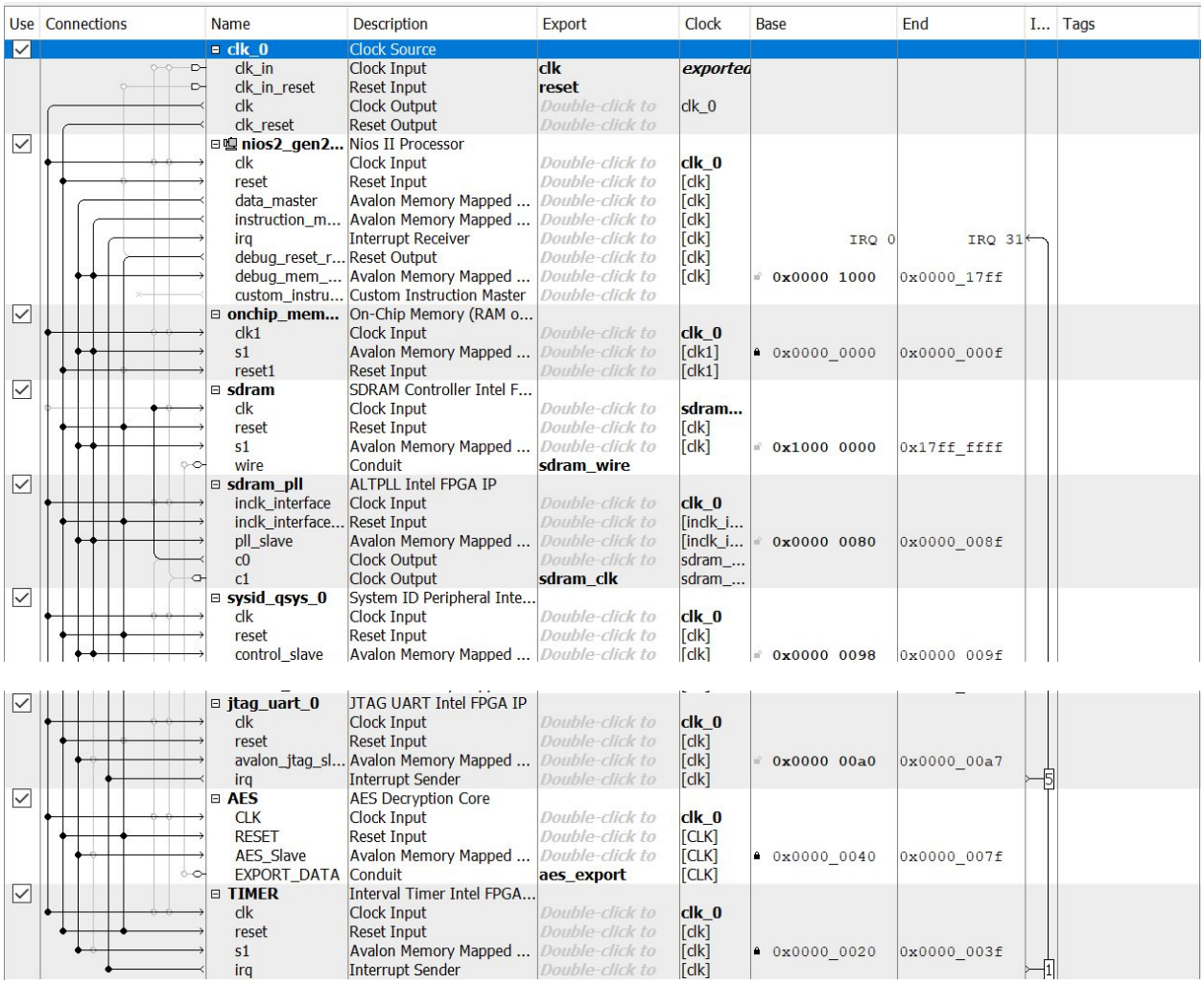

Figure 3.1: QSYS (Platform Designer)

Annotated Simulations of AES Decryption

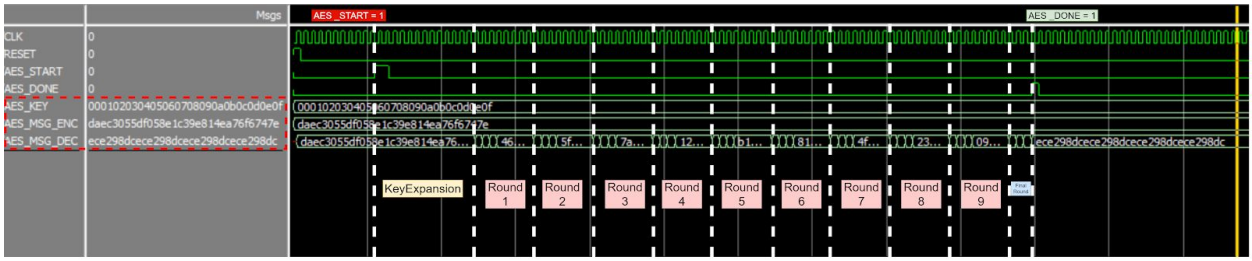

Figure 4.1: AES Module Simulation (Annotated)

Post-Lab

LUT	5,746
DSP	None
Memory	571,392
Flip-Flop	2,886
Frequency (MHz)	52.54
Static Power (mW)	102.53
Dynamic Power (mW)	75.62
Total Power (mW)	245.58

Figure 5.1: Resources and Statistics

We both expected hardware to be faster partly due to the inherent slow speed of the NIOS II but mainly due to the sequential nature of the software, where one computation is achieved at a time in each each step of the encryption process. On the other hand, in hardware a lot of the computations are done in parallel and the current state in the state machine determines which function output is going to be the next state of the current decrypted message. Our speed for hardware was ~200 KB/s whereas the software was ~0.55 KB/s which confirms our initial expectations.

An improvement to the hardware speed could have been to instantiate 4 inverse mix columns in order to make the operation work in parallel as opposed to waiting 4 additional clock

cycles for inverse mix columns to complete the operation on the state of the message. Generally speaking, operations that take multiple cycles to complete that don't rely on previous intermediate results should be done in parallel to speed up the process.

Conclusion

A majority of the problems in this lab came from the hardware approach (week 2). Things like our code not synthesizing the things we want it to synthesize required us to take different approaches to the decryption code. We also had to switch FSM implementations to improve performance (i.e. instead of ~90 explicit states, we compressed it to 12 declared states and just looped, similar to the software approach). In addition to that optimization, we had to optimize our decryption process by removing unnecessary for-loops inside of the decryption function in main.c (explicit declarations vs. using for loops to initialize variables affects performance). While those were some of the big problems, most of our tedious problems were simple typos or placing code in the wrong blocks (i.e. invalid begin/end structures). The only other problems we had was not understanding how to implement mix columns in week 1 because documentation on that wasn't really clear. Everything else was relatively straightforward.

This lab did well in teaching the difference between hardware and software performance on complex processes like AES. While software may be more straightforward when it comes to implementation, this straightforwardness comes at a cost which is performance. Optimizing software requires writing parallel performing operations which aren't as intuitive as the hardware approach.