Microprocessor-Based Systems (E155)

Lab 7: Advanced Encryption Standard

Learning Objectives

By the end of this lab you will have…

- Learned to read and implement a complex specification
- Built a nontrivial system on an FPGA that requires thoughtful architecture to fit on the chip
- Designed and implemented an interface to communicate between the FPGA and microprocessor on your µMudd board
- Learned how to use a logic analyzer to analyze and debug your system
- Gained experience with hardware accelerators

Requirements

Construct a hardware accelerator to perform 128-bit AES encryption. Send a plaintext message and key from a microprocessor to the accelerator and verify that the cyphertext received back is correct. Display the SPI communication on the logic analyzer.

Advanced Encryption Standard

The Advanced Encryption Standard is described in an unusually succinct and clear standard. Reading the standard carefully will save you time. See Appendix A-1 for an example of the key expansion during each round and Appendix B for an example of the intermediate results during each round.

Implementation

Download and unzip SAM4S4B_lab7 and place it in the same directory with the rest of your code. This is an expanded version of the SAM4S4B_lab6 with support for more peripherals.

Download aes_starter.sv, sbox.txt, and lab7.c from the web page. aes_starter.sv contains the top-level module, an SPI interface, and a testbench that applies and checks the test vector described in Appendix A-1 and B. It also contains the mixcolumns logic that operates on a 128-bit intermediate state. The Galois field arithmetic for mixcolumns is more complicated than for the rest of AES, and the implementation is based on a paper cited in the code. The sbox module and sbox.txt lookup table perform the sbox substitution on a single byte. Lab7.c sends a key and plaintext message over SPI to the FPGA, then checks that the result is correct. Set up your project in Keil using the same settings as in Lab 5, and remember to make the same modification to ‘startup_SAM4S.c’.

You will discover that the logic is too large to implement all the rounds as one giant block of combinational logic. Therefore, you will need to perform the rounds sequentially.

Turn in the usual report including design approach, block diagram, code, schematics, results, and time spent.

Hints

Previous students have spent a highly variable amount of time on this lab. Here are some suggestions to make it go faster.

Start by thoroughly understanding the spec.

In prior labs you may have gotten in the habit of thinking in code. Remember to go back to thinking about hardware rather than function calls. Draw a block diagram for your hardware using elements such as registers, multiplexers, FSMs, and blocks of combinational logic. Name all of the signals between blocks. Remember how the E85 multicycle processor had a datapath that required certain control signals such as mux selects, and a controller that generates the control signals at the appropriate times. You’ll find a similar organization helpful. Write idiomatic Verilog code that exactly matches your block diagram.

Watch for warnings in synthesis and simulation, and correct these before moving ahead.

Get your design working in simulation first. When debugging, find the first place you can tell a signal is wrong. Add all the relevant inputs that influence that signal. If one of them is wrong, recursively work backward. When the inputs are good and the output is bad, you’ve isolated the bug and can look for it in that part of the code. Learn to do this systematically so you can find and solve each bug in minutes rather than hours.

If the design works in simulation but not on hardware, it is often a wiring error or a discrepancy between how you timed your control signals and what the C code expects. Make sure you’ve read the provided code carefully and are producing signals at the right times. Check that your FPGA and microcontroller are expecting the same polarity and phase for your SPI clock signal. Use the many channels of the logic analyzer to view all of the relevant signals at once and check them against your expectations. If you have a hard bug, it’s helpful to tap out intermediate signals, such as the state of a FSM, onto FPGA pins so you can watch them on the logic analyzer.
Have fun! This is a sophisticated system and you should feel proud when you have built and debugged it!

Credits

This lab was original developed in 2015 by Ben Chasnov ’16 and redesigned for the µMudd Mark 5.1 in 2019 by Caleb Norfleet ’21.