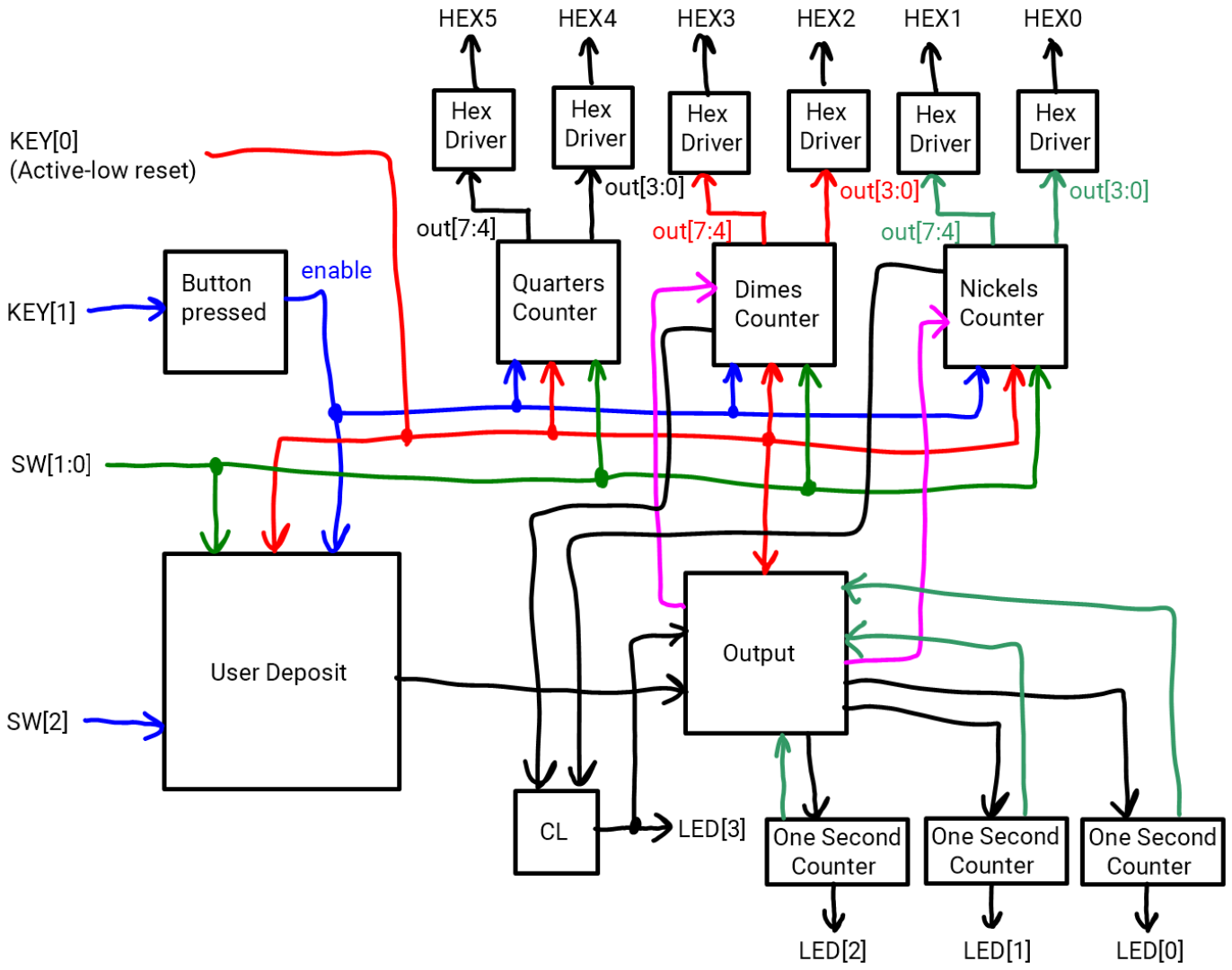
Contents

Project Objectives

Project 4 called for the design and construction of a finite state system to emulate a vending machine. This vending machine takes in quarters, dimes, and nickels, and dispenses a product when 60 cents have been inserted by the user. If change is available in the vending machine, change will be given back for input values above 60 cents. This machine also has a “maintenance mode” that allows coins to be inserted into the vending machine without counting towards the 60 cents to purchase an item.

This vending machine keeps track of how many coins are in it via six 7-segment displays. Four LEDs are used to represent whether exact change must be given to the machine (IE change will not be given to the user if they go over 60 cents), a product is being dispensed, a dime is being given back in change, or a nickel is given back as change. In the case of coins being given back as change, the vending machines inventory of that coin will be decremented.

Design and Implementation Process



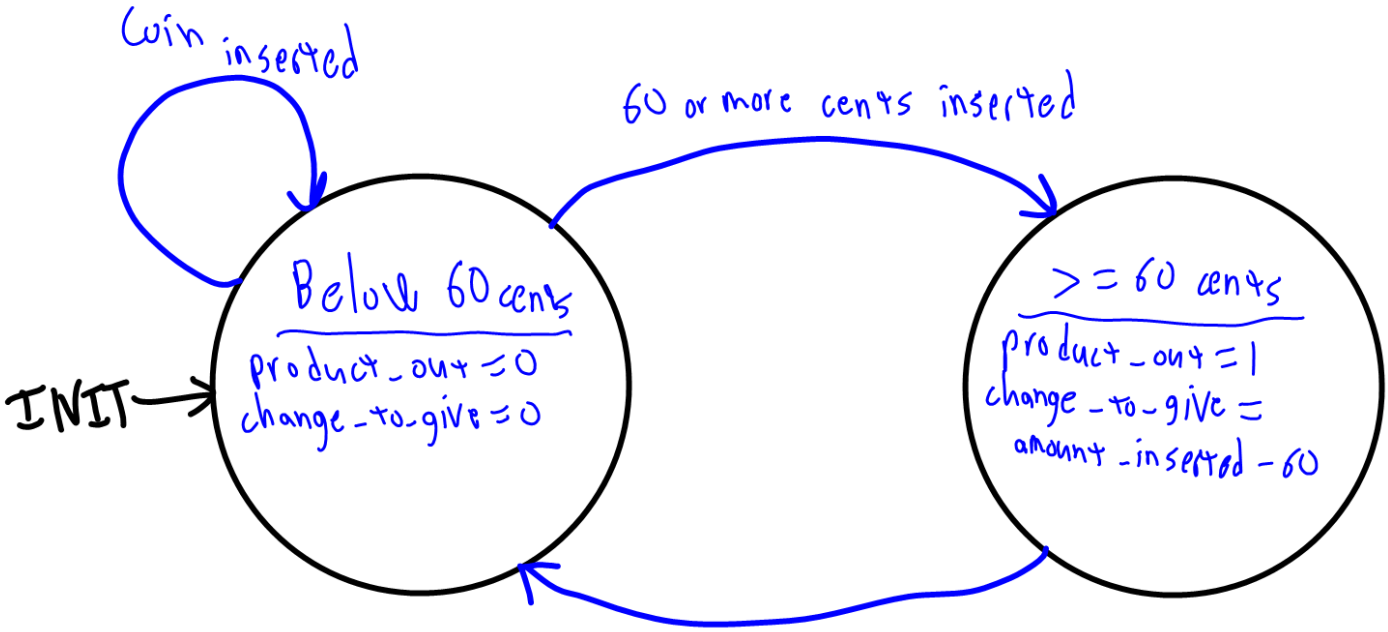
My design process began with generating the block diagram above. Firstly, the machine needed counters to keep track of how many of a given coin it has. These counters then output their “contents” into two hex driver modules to handle the output to the 7-seven displays. They also needed to increment and decrement inputs.

From here, I decided to make a “user deposit” module that would keep track of how much money had been deposited by the user. When enable is asserted and a positive clock edge occurs, this module will add 25, 10, or 5 cents to its counter according to the coin that was inserted. This input is ignored if the system is in maintenance mode. Upon reaching 60 or more cents in the system, a product dispense signal and the amount of change to be returned to the user is sent to another module: the output control module.

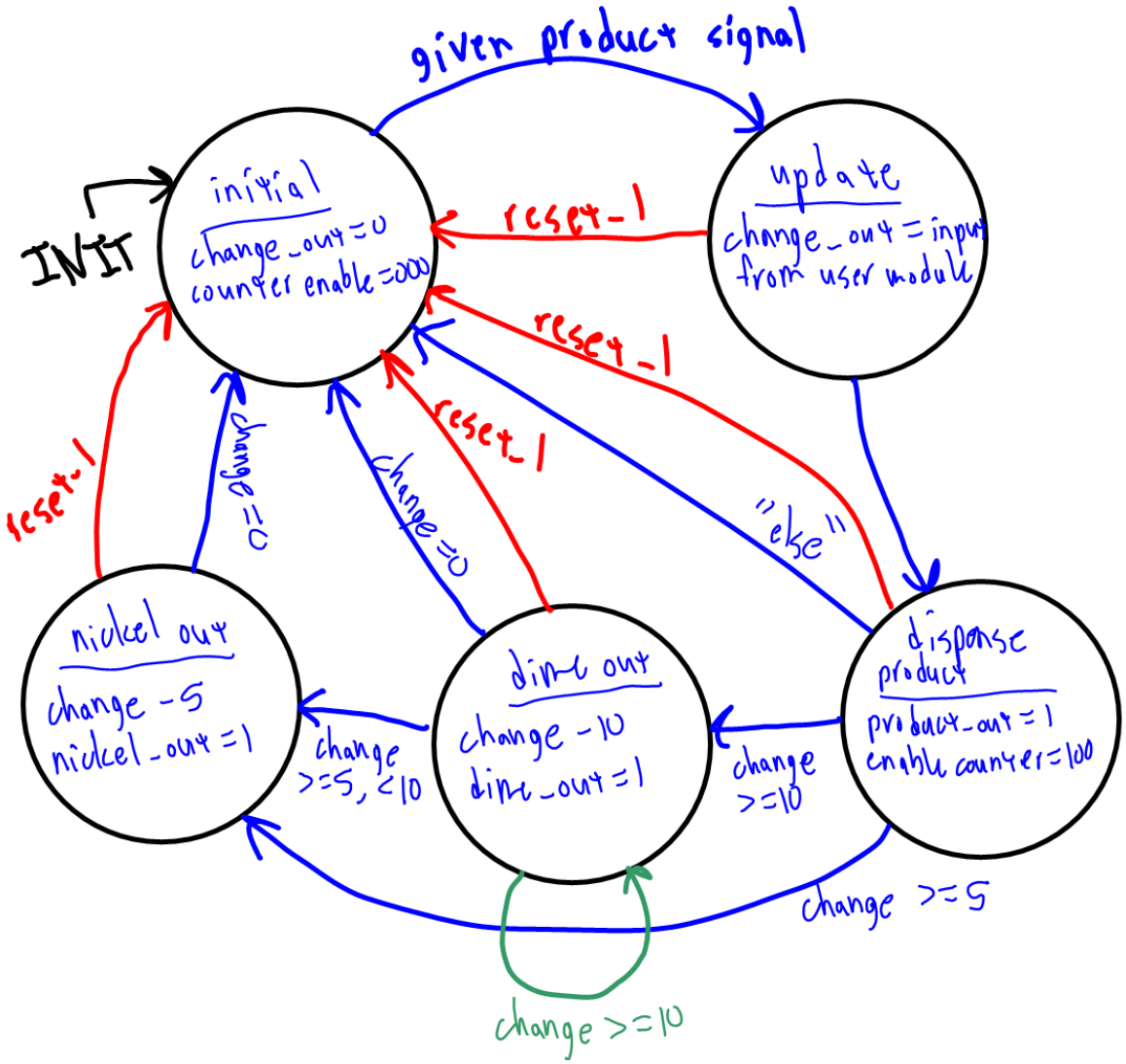
This last module, dubbed output control, does exactly as its name implies. It takes the product dispense signal and the value sent from the user deposit module and a “exact change” signal as inputs to determine the correct outputs and their order. It sends enable signals to start a 1 second one shot timer for a given output (Product out, dime out, nickel out). It also sends decrement signals to the coin counters as change is given back to the user.

Once the block diagram was complete, basic state diagrams were created to have a rough blueprint out of how to build each individual module before wiring them together. After building all these modules individually, they were wired together in the project4 module. Simple tests were run as components were added to the system to make sure each new module added worked correctly. As expected, there were errors as modules were added. As the pieces were being added one by one, errors were easier to debug and correct.

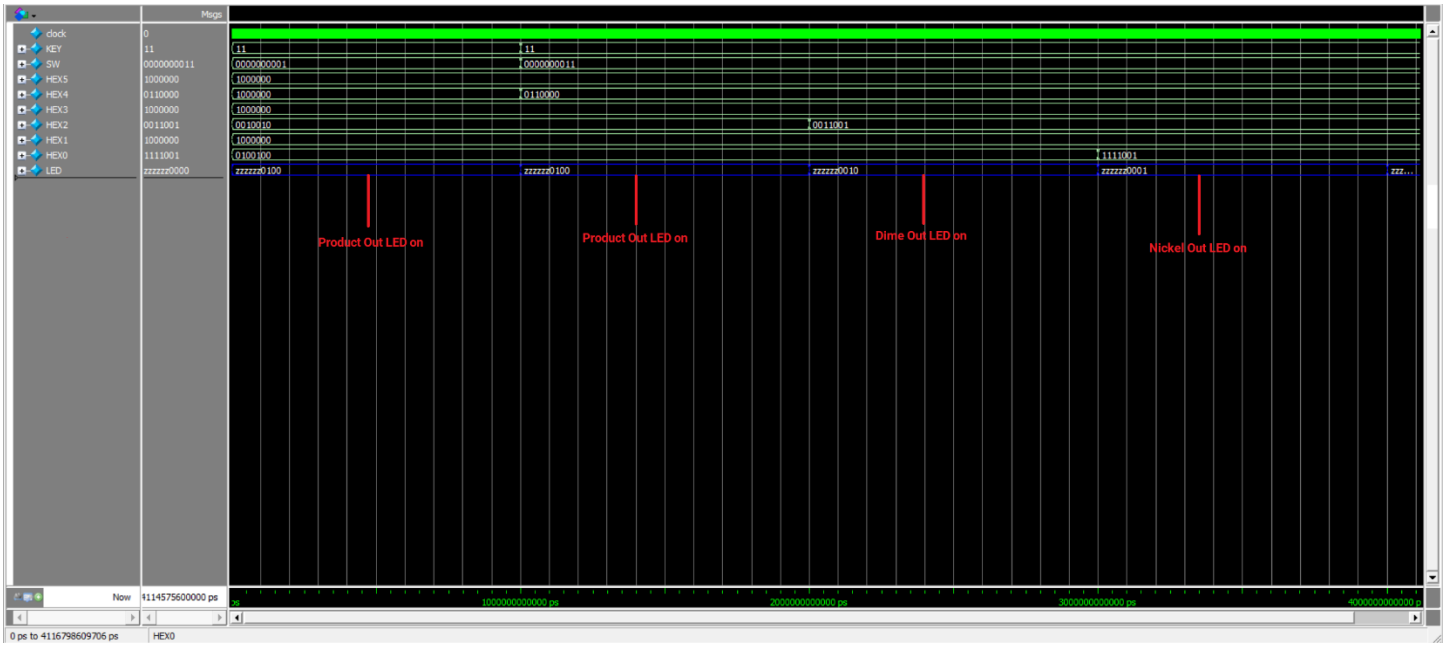
Finally, the entire system was instantiated and compiled. A test bench module was made to run a few simple cases to test the full system. The results of these tests are shown below.



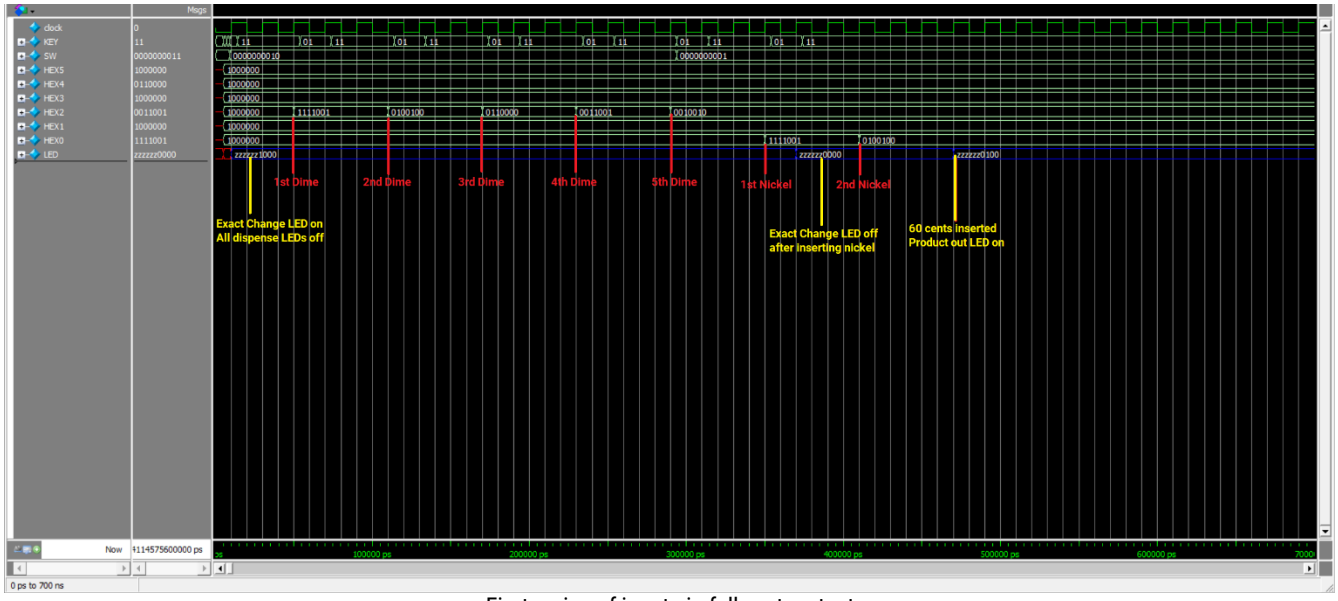
State Diagram of User Deposit module



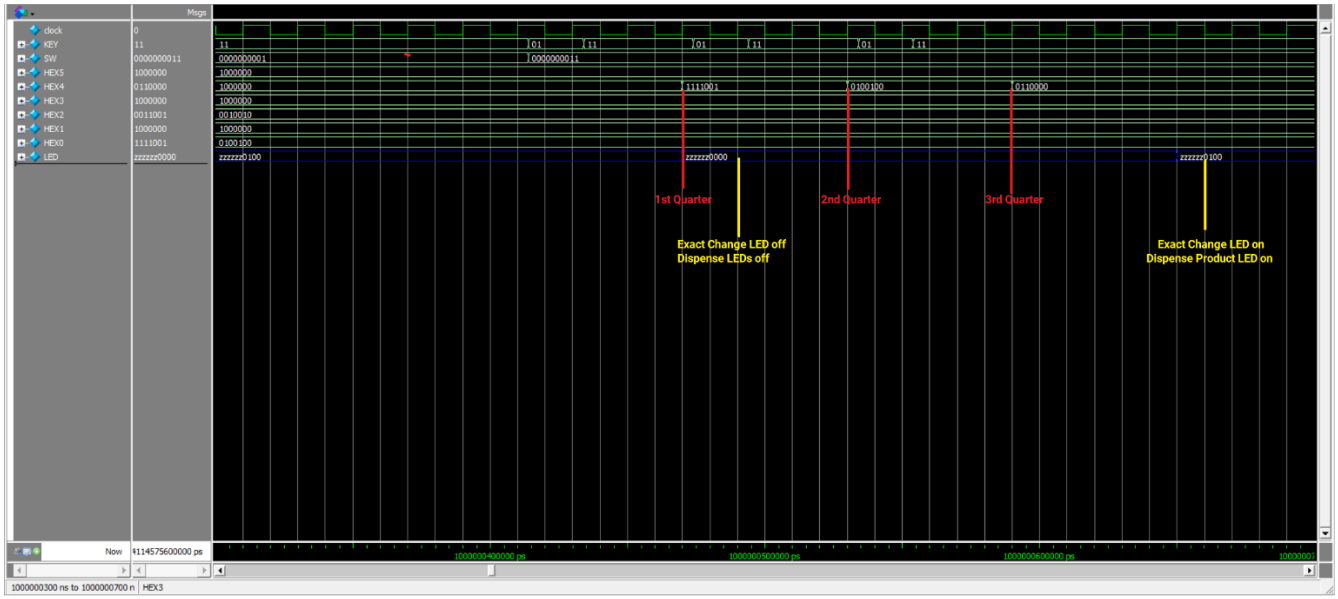
State Diagram of Output Control module



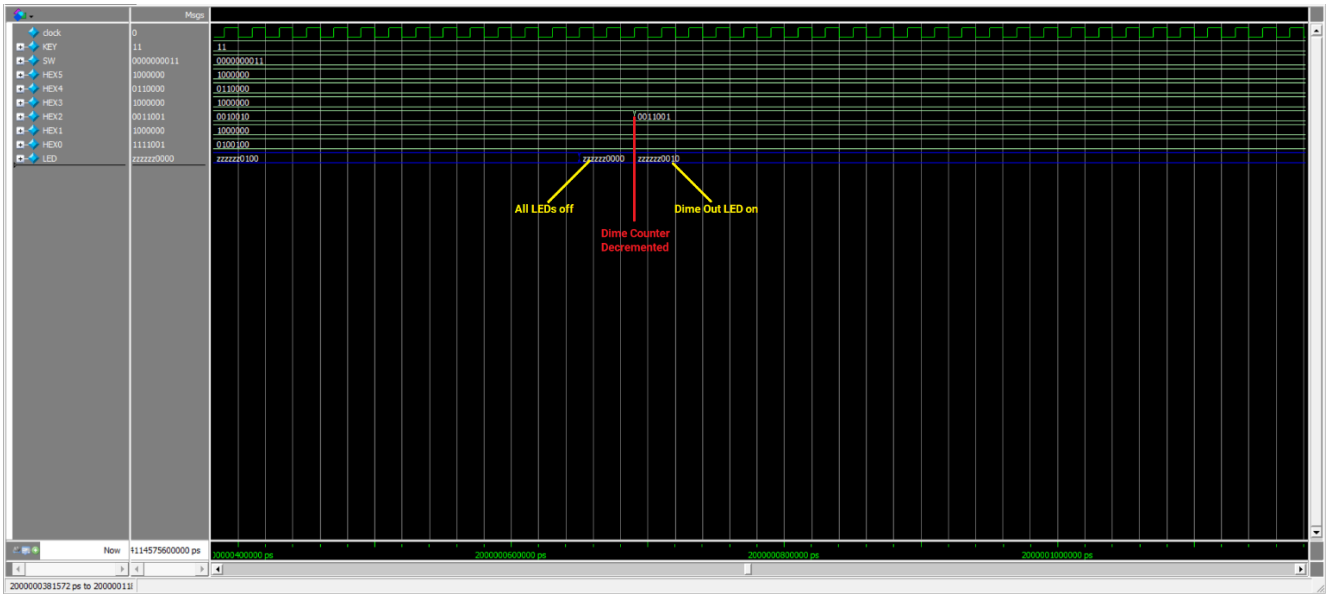
Full View of the full system test bench



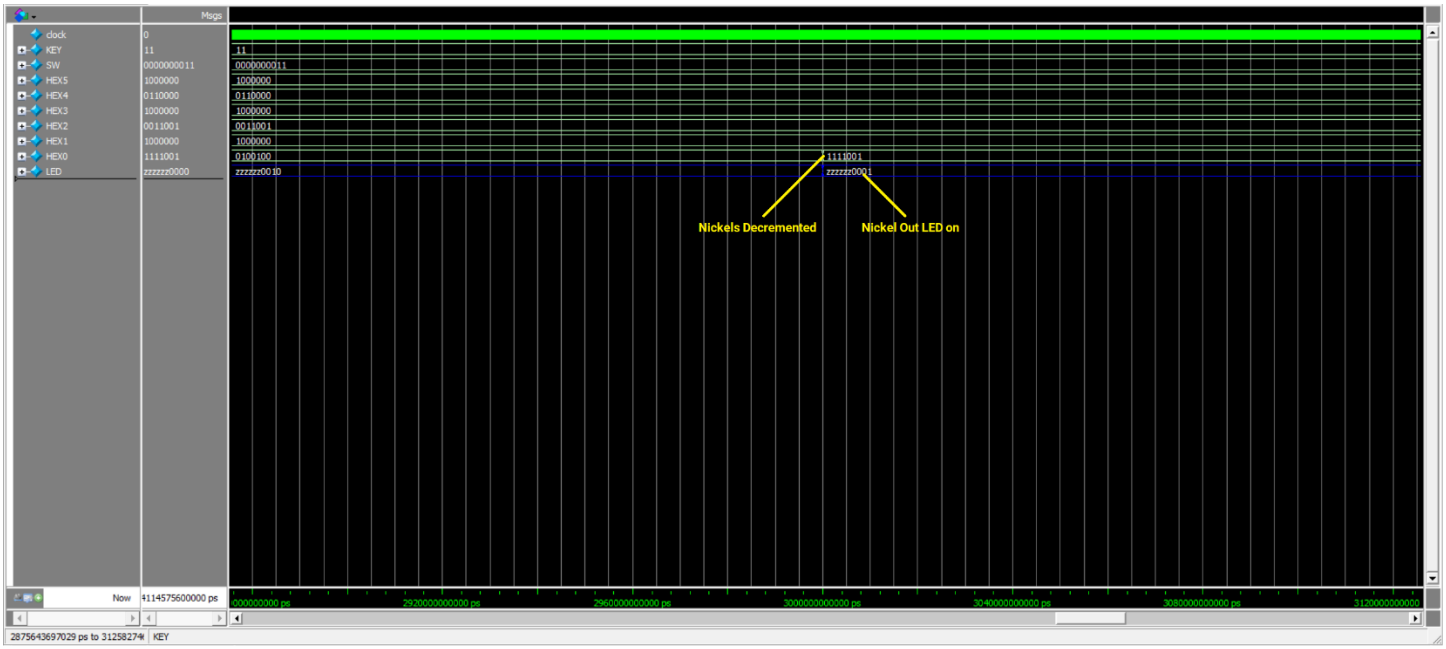
First series of inputs in full system test



Second series of inputs in full system test



Correct function of Output LED to Dime Out LED



Correct function of Dime out LED to Nickel Out LED

After these tests showed some correct function, the project4 module was flashed to the FPGA. From here, much more exhaustive testing was done to account for most of the common inputs to the system. The FPGA passed this more extensive testing and I feel confident in saying my system works as intended.

Conclusion

Overall, this project was a fun start-to-finish project. I enjoyed being able to see a full system be developed from a specification to a working solution on the FPGA versus the “here’s some starter code, fill in the blanks” projects we get sometimes. Proper development techniques (designing the block-level diagram of the system and then the rough state machines of each major module before writing any code) made the process relatively painless.

I say “relatively” because while the high-level design was very simple, the coding part wasn’t as simple. I made a few mistakes in the always@ blocks that took me a while to figure out, but I learned a few important lessons about how to structure those blocks in the future. I also learned that debugging Verilog systems is far less intuitive than “conventional” coding. Instead of a quick cout statement in C++, an entire testbench must be designed and compiled to test the system and then the waveform results have to be deciphered to identify what’s going wrong in the system and how to fix it.

Ultimately, I got a lot of practice writing multiple state machines that have to work with each other. Doing so taught me a couple of bad habits I need to break and the practice is helping Verilog/hardware states (as opposed to C/software states) become more intuitive to me and to make better sense of the intricacies of both.