Contents

Design Specification

Project 5 required the design of a processing unit that could handle several different operations. These operations had a specified 4-bit opcode and took a differing number of 8-bit operands and output a 16-bit value that drove four 7-segment displays. Opcodes and operands were input using switches on the FPGA.

The operations supported by this unit are specified in Table 1-1. Most operations took place over 3 cycles (load opcode, load Operand A, load operand B). Two operations took 2 cycles (1's Compliment, Negate), while multiplication took 8. Some functions involved some data manipulation (sign extending operands) prior to performing the functions. Most of these functions were done via dataflow operators with the exception of shifts and multiplication. Shifts were done via concatenation and multiplication was done using a separate multi-cycle module designed in Homework 7.

Opcode	Operation	Description
0000	AND ID	Bitwise AND of {A, B} and last four digits of ID (7383) as BCD
0001	NAND	Bitwise NAND operands A and B
0010	NOR	Bitwise NOR operands A and B
0011	XOR	Bitwise XOR operands A and B
0100	1's Complement	Bitwise complement of operand A
0101	Add	Add operands A and B. Operands are 8-bit signed 2's complement, sign-extended to 16-bits
0110	Subtract	Subtract operand B from operand A. Operands are 8-bit signed 2's complement, sign-extended to 16-bits
0111	Negate	2's Complement of operand A. A is an 8-bit signed 2's complement, sign-extended to 16 bits
1000	Arithmetic Shift Left	Arithmetically shift operand A to the left by B positions. B is a 4-bit unsigned value
1001	Arithmetic Shift Right	Arithmetically shift operand A to the right by B positions. B is a 4-bit unsigned value
1010	Rotate Left	Circular Shift operand A to the left by B positions. B is a 4-bit unsigned value
1011	Rotate Right	Circular Shift operand A to the right by B positions. B is a 4-bit unsigned value
1100	Multiply	Multiply operands A and B. A is the multiplicand, and B is the multiplier.

Table 1-1 – List of functions

Design Process and Implementation

After reading the design specification, a system level state diagram was made (Figure 1-1) and a basic block diagram of the hardware (Figures 1-2, 1-3) was also made. These were left to be general guides, as it was determined that the specifics would likely be changed when writing the Verilog.

Firstly, modifications were made to modules that were made in previous assignments. These modules performed exactly as needed for this project, but the bus sizes needed to be expanded from 4-bits to 8 and extra control I/O needed to be added. After that was done, work began on the ALU Core module first. It was decided that it'd be easier to revolve the specifics of the control unit around the function module rather than conforming the function module to the control module.

The ALU Core was straightforward to build. Time was spent trying to create a more elegant solution for the shift operations than a series of case structures, but ultimately that is what was implemented, as any other cleaner or more elegant solution would've been more time consuming and error prone to write. After this module was built, a very simple test bench was written to confirm the basic function of loading the correct registers in the correct order and that the first few functions worked correctly. The resulting waveform (Figure 2-1) confirmed that AND ID, NAND, NOR, and ADD all worked. With the ALU Core module loading in values correctly, work began on the ALU Control module. This module created the state diagram in Figure 1-1 and asserted control signals as needed.

With all the components built, all the modules were instantiated in the top-level project 5 module. Another testbench was written to test the system and to aid in the inevitable debugging process. After adding debug signals and ports, an addition case of 3 + 2 was run on the system. This testbench led to fixing several timing issues and ultimately showed correctly executed addition, 1's complement, and AND ID operations.

Finally, the full system was flashed to the FPGA and every function was tested with several examples that were worked out by hand. A timing issue was found once again and was fixed. The multiplication function had a similar timing issue and that was again fixed. This issue will be discussed further in the conclusion. Two states (Wait and Complete) were also eliminated in the final design.

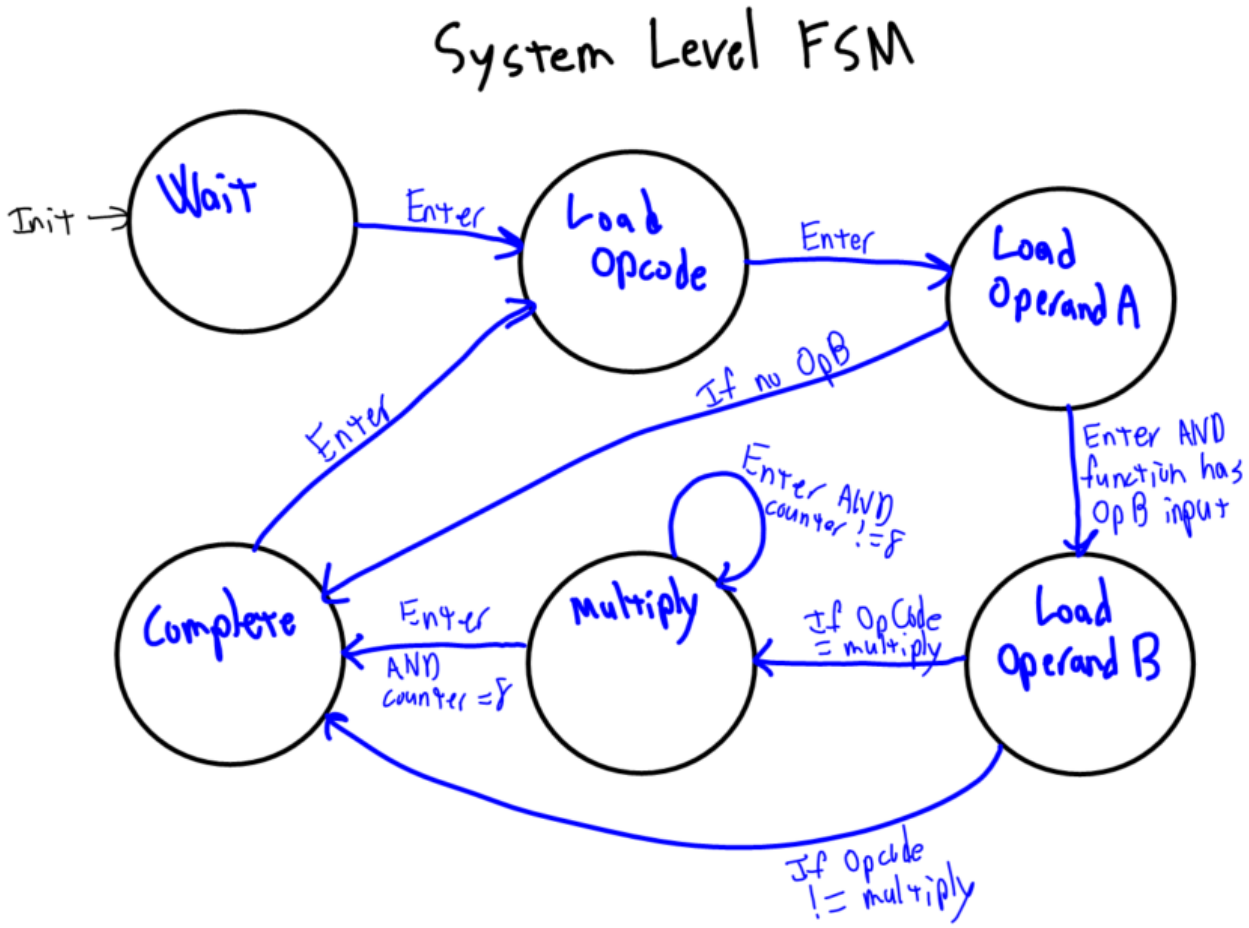

Figure 1-1 – Initial System Level State Diagram

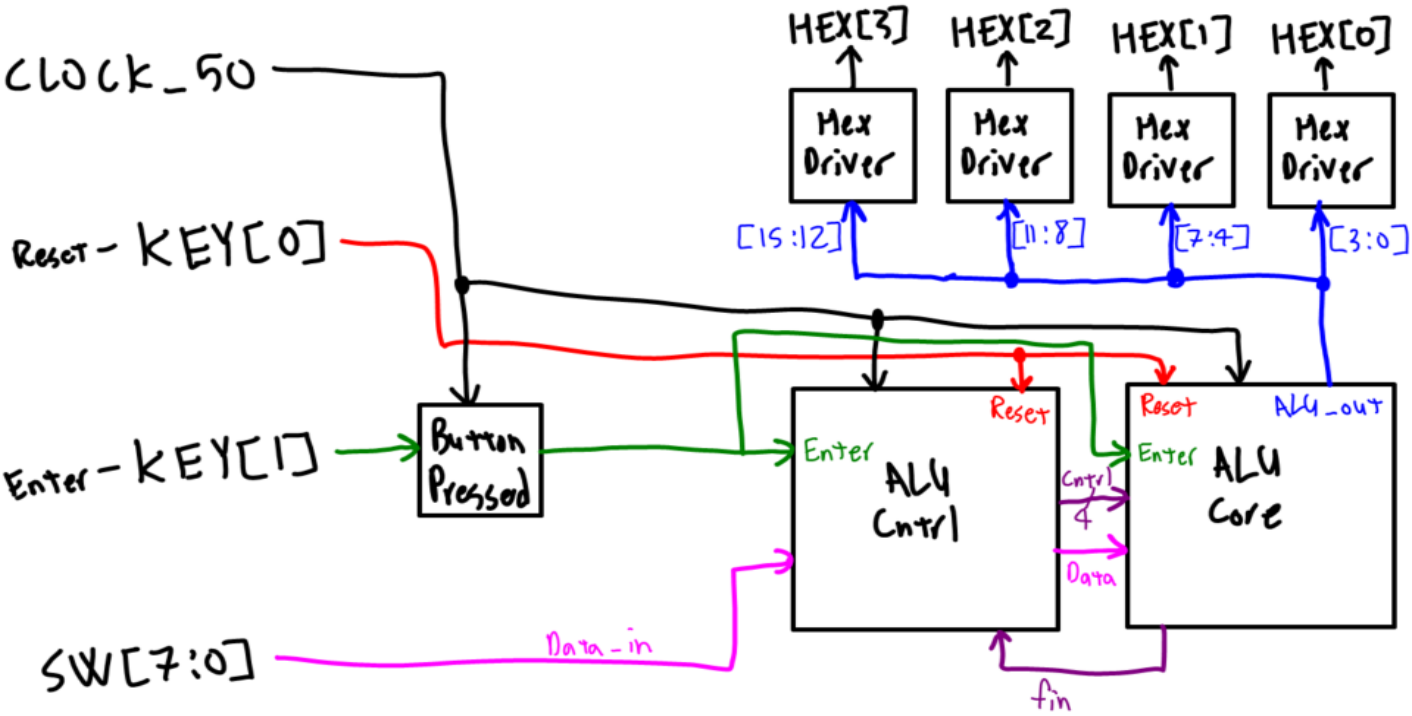

Figure 1-2 – System Level Block Diagram

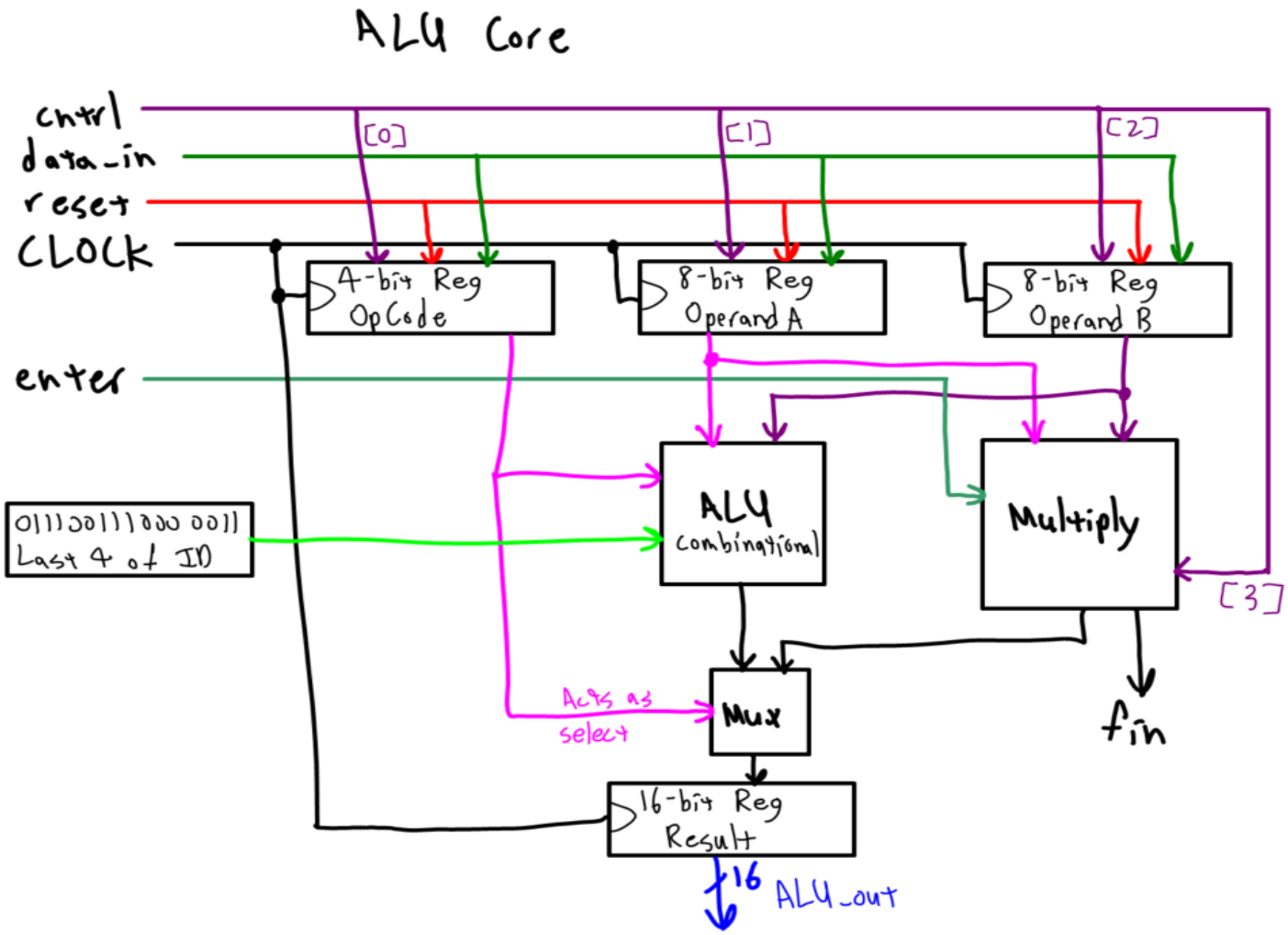

Figure 1-3 – Initial Block Diagram of ALU Core

Test Results

Initial ALU Test

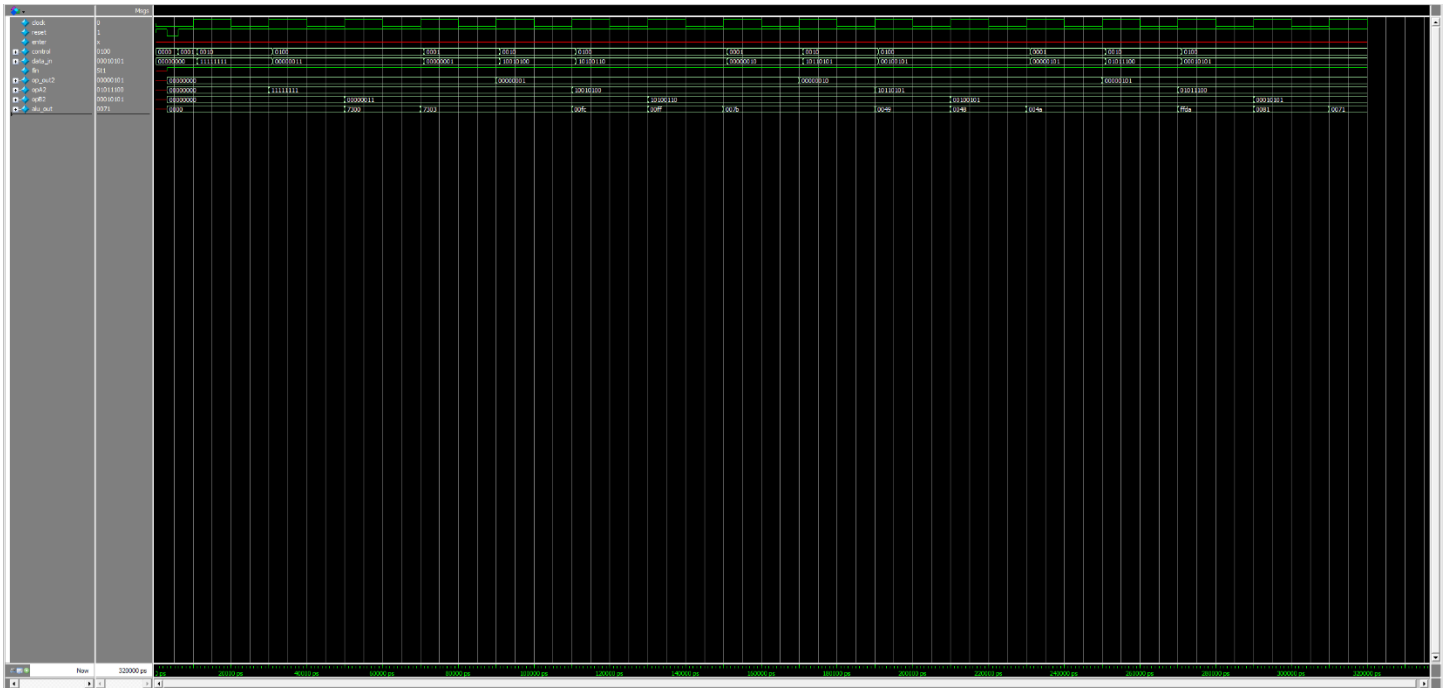

Figure 1-4 – Simple testbench output waveform

Full System Test Results

Note: these tests are all in commented out blocks inside tb_project5_aep277.v. Simply uncomment them to run a particular test.

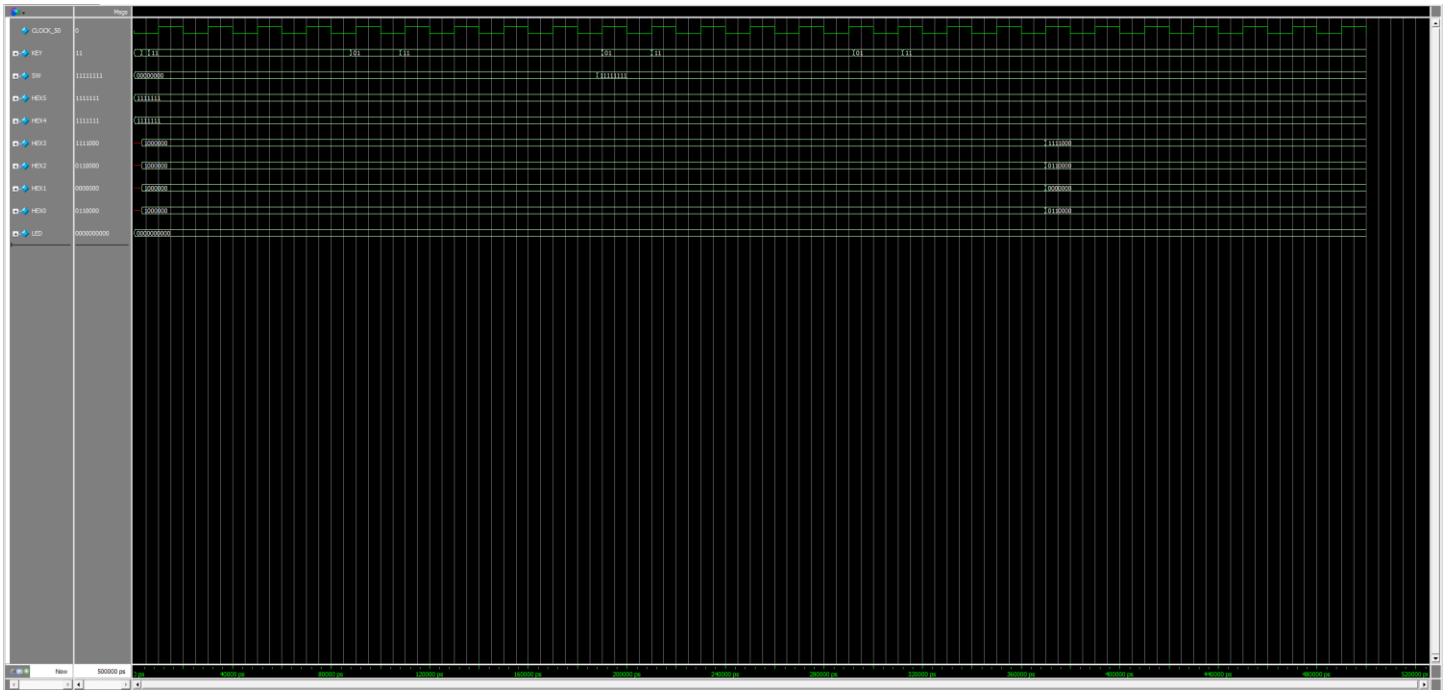

Figure 2-1 – AND ID

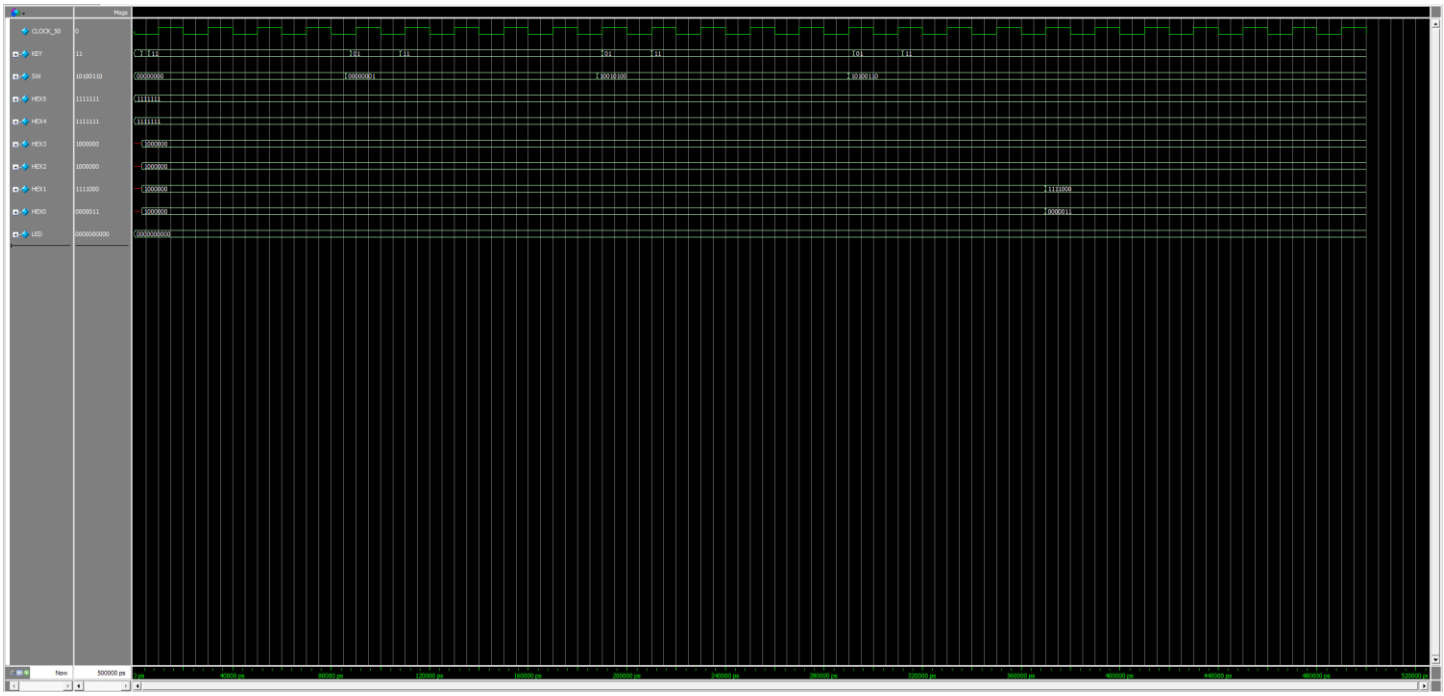

Figure 2-2 – NAND

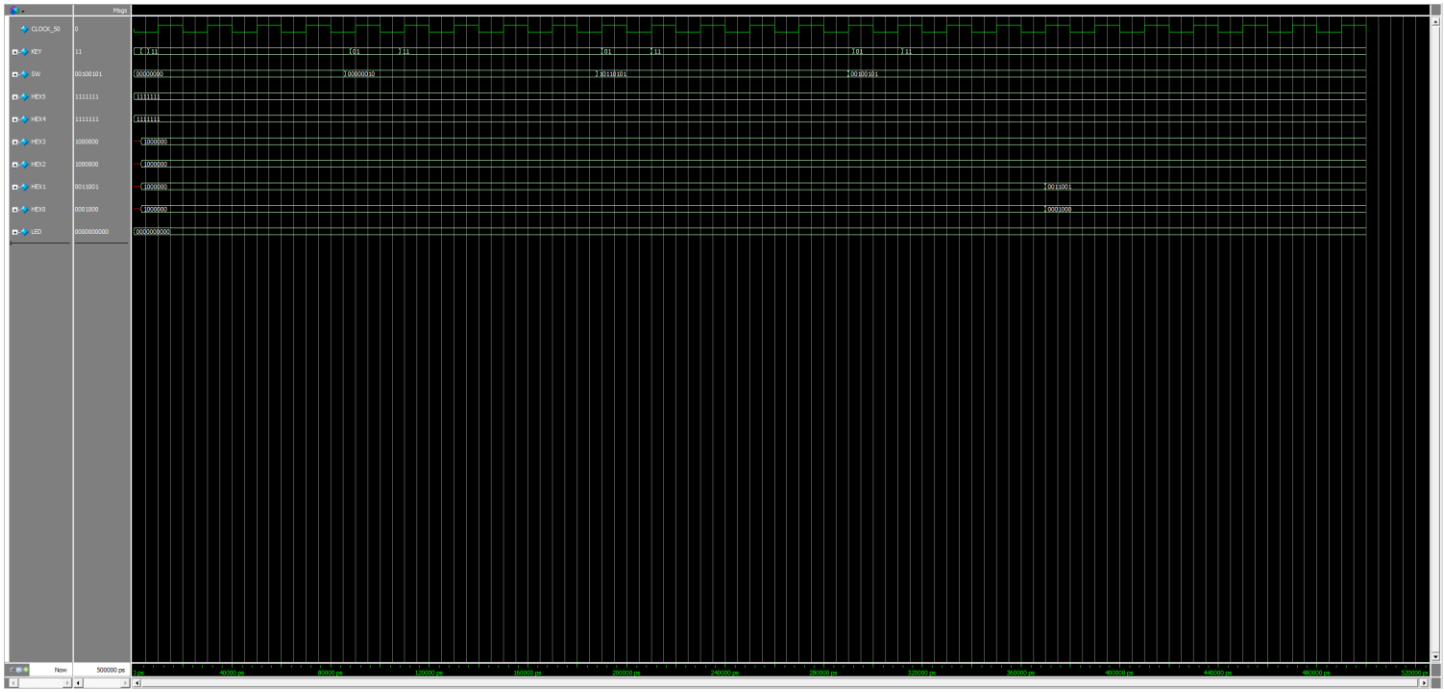

Figure 2-3 – NOR

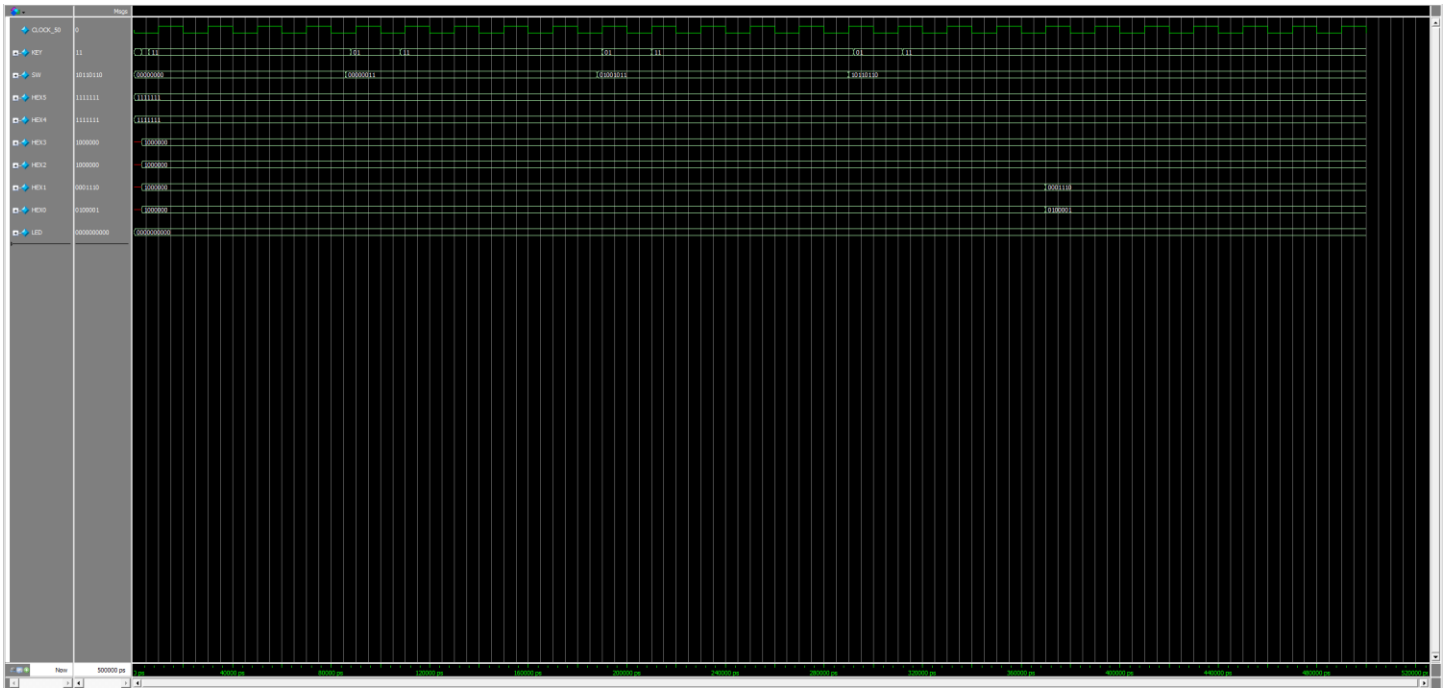

Figure 2-4 – XOR

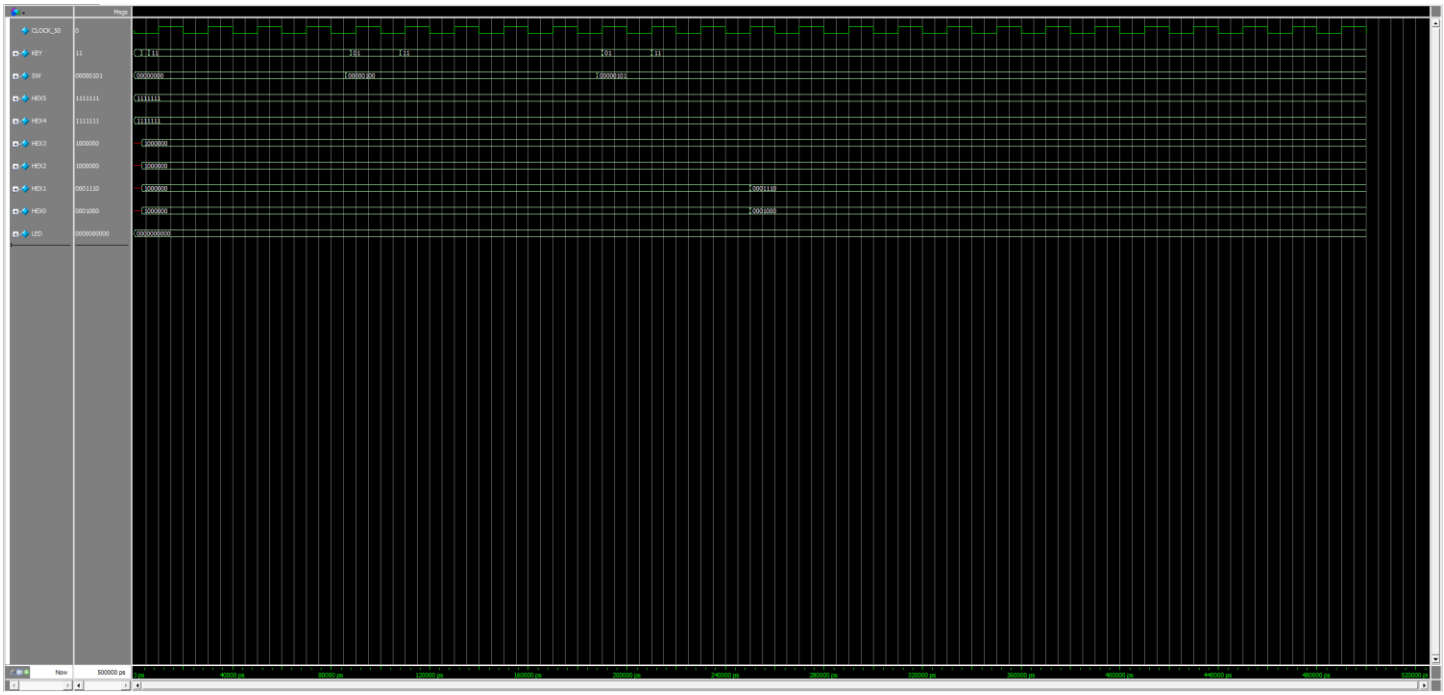

Figure 2-5 – 1's Complement

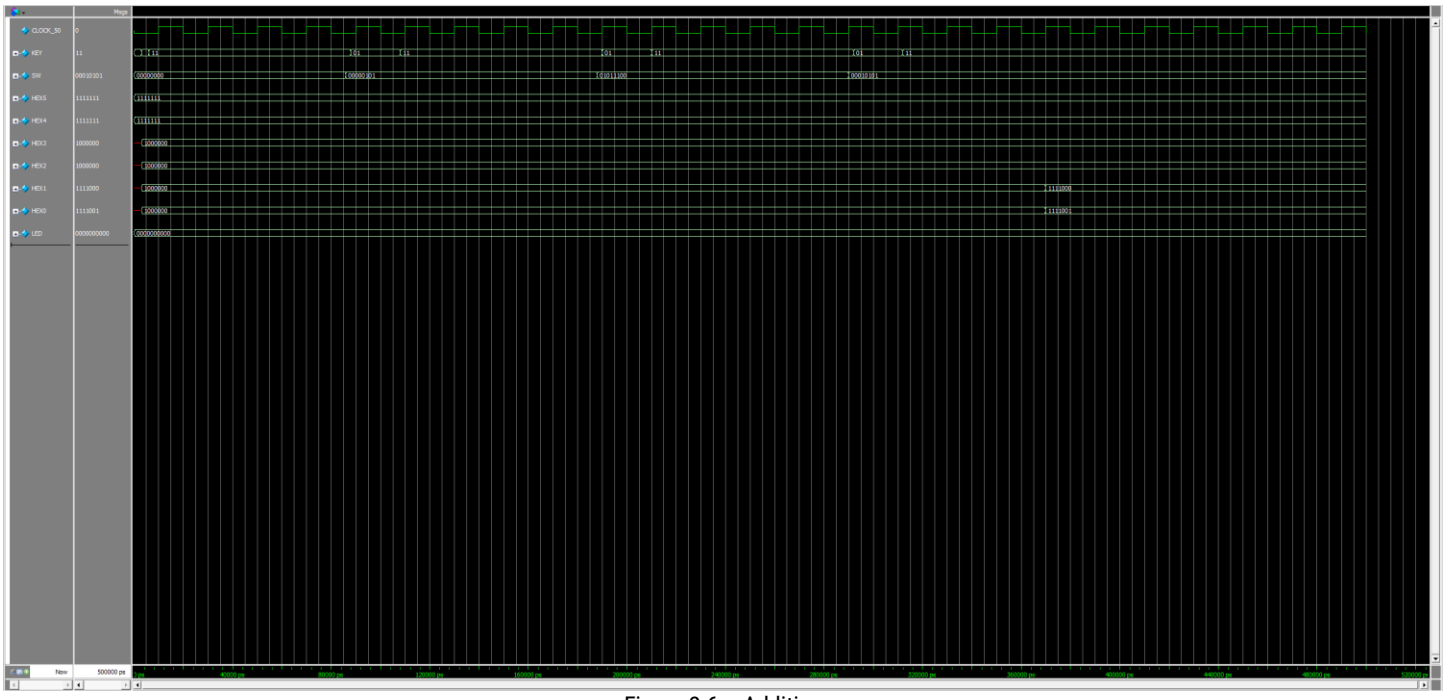

Figure 2-6 – Addition

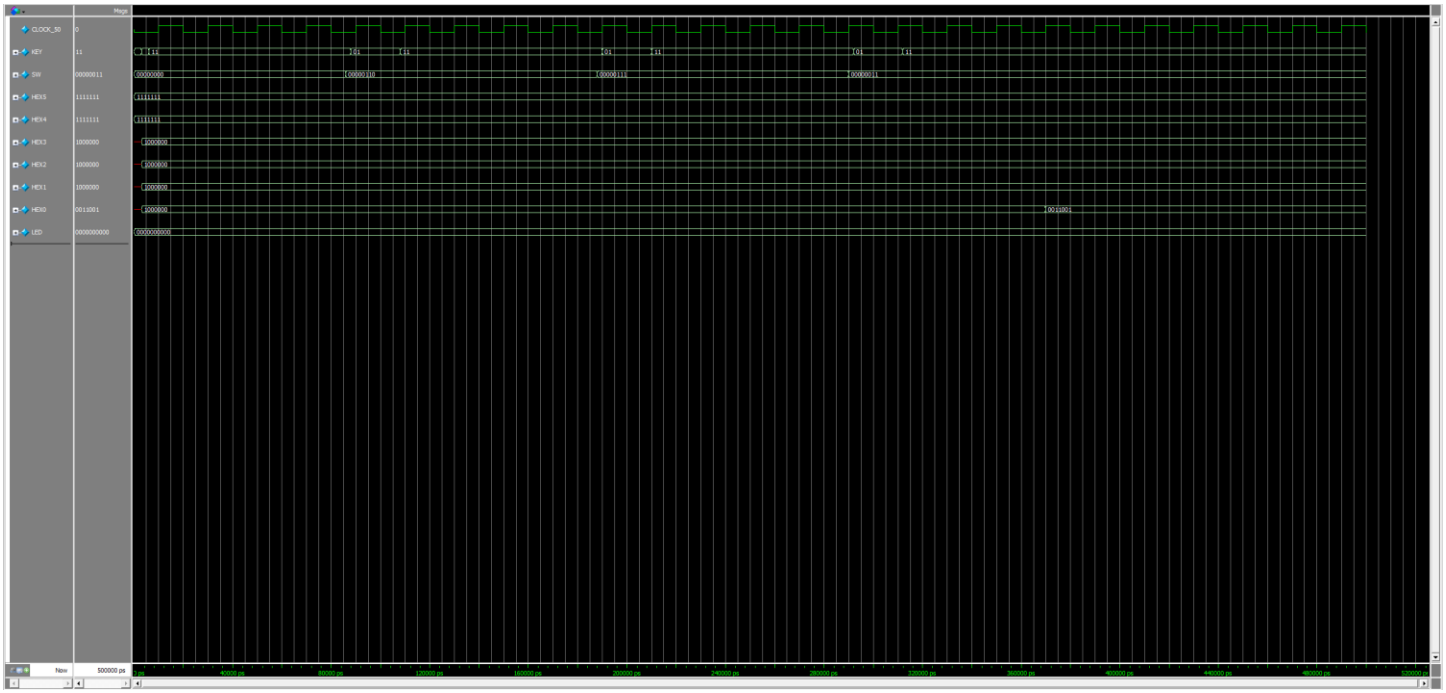

Figure 2-7 – Subtraction

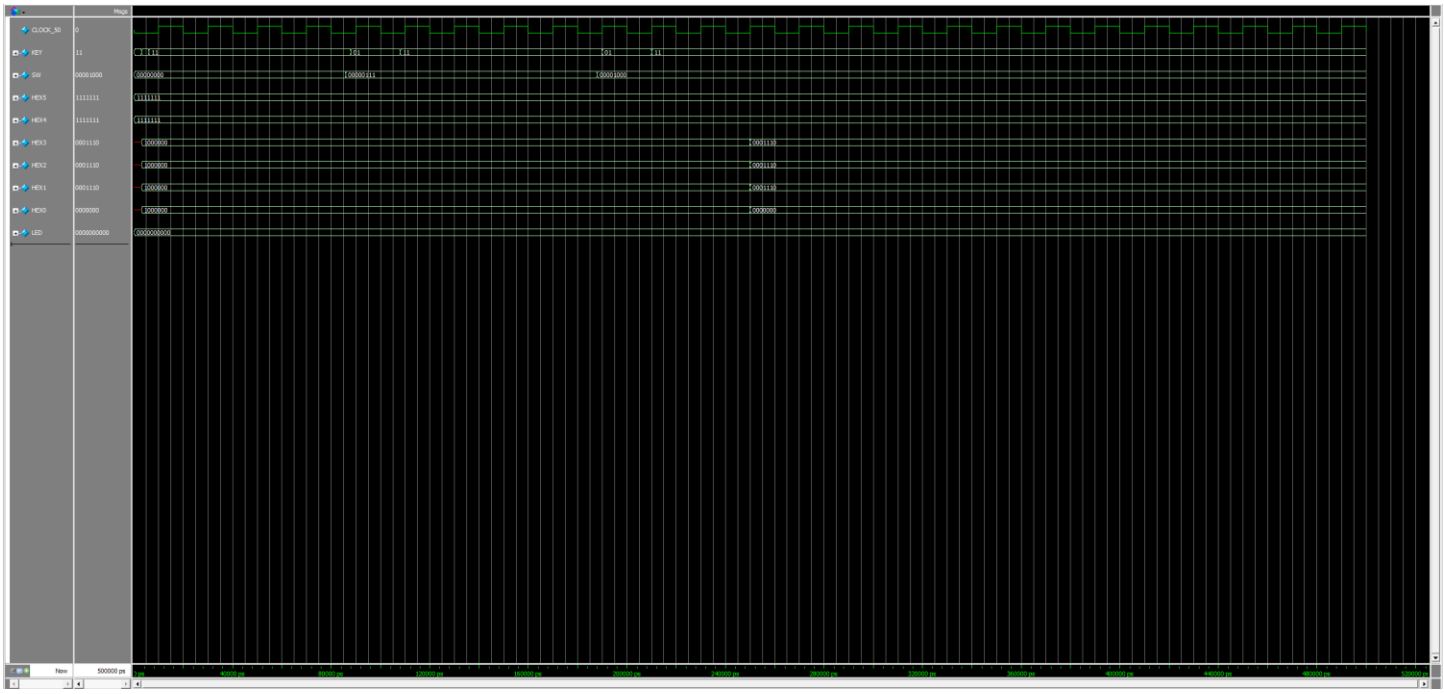

Figure 2-8 – Negate

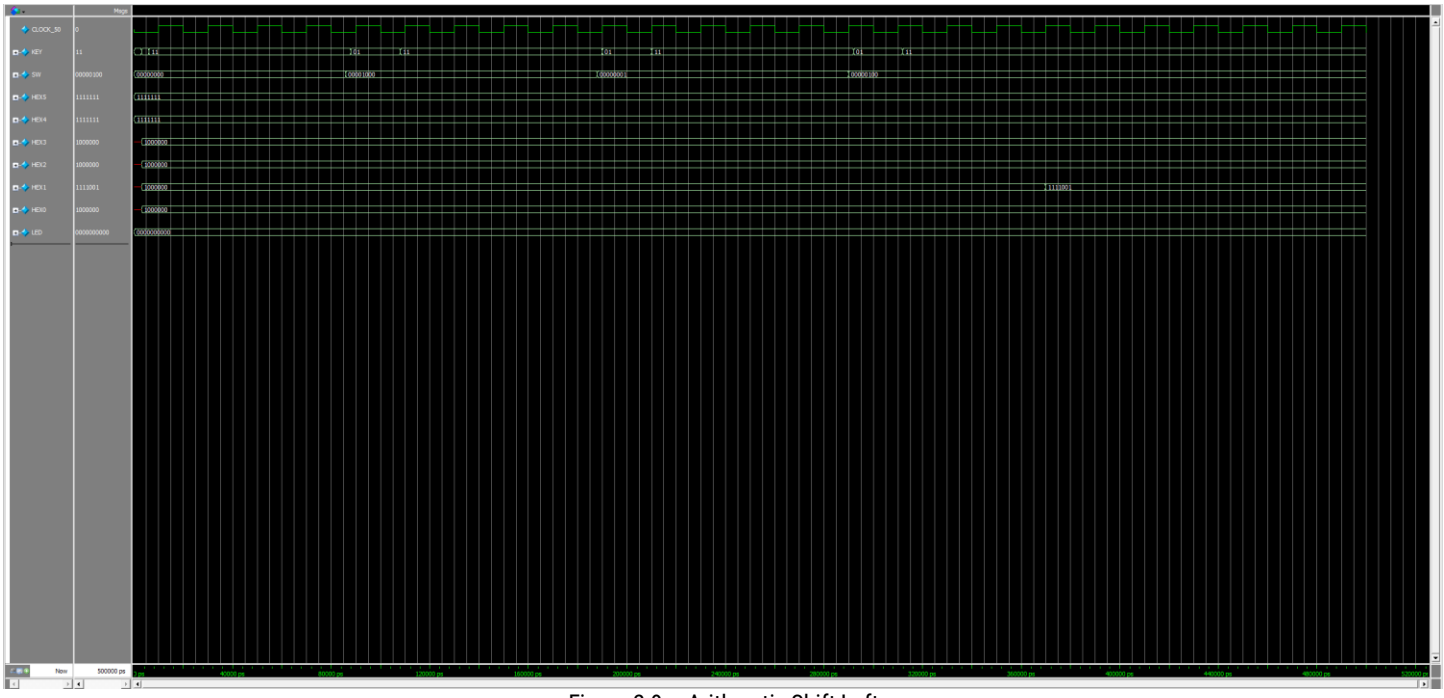

Figure 2-9 – Arithmetic Shift Left

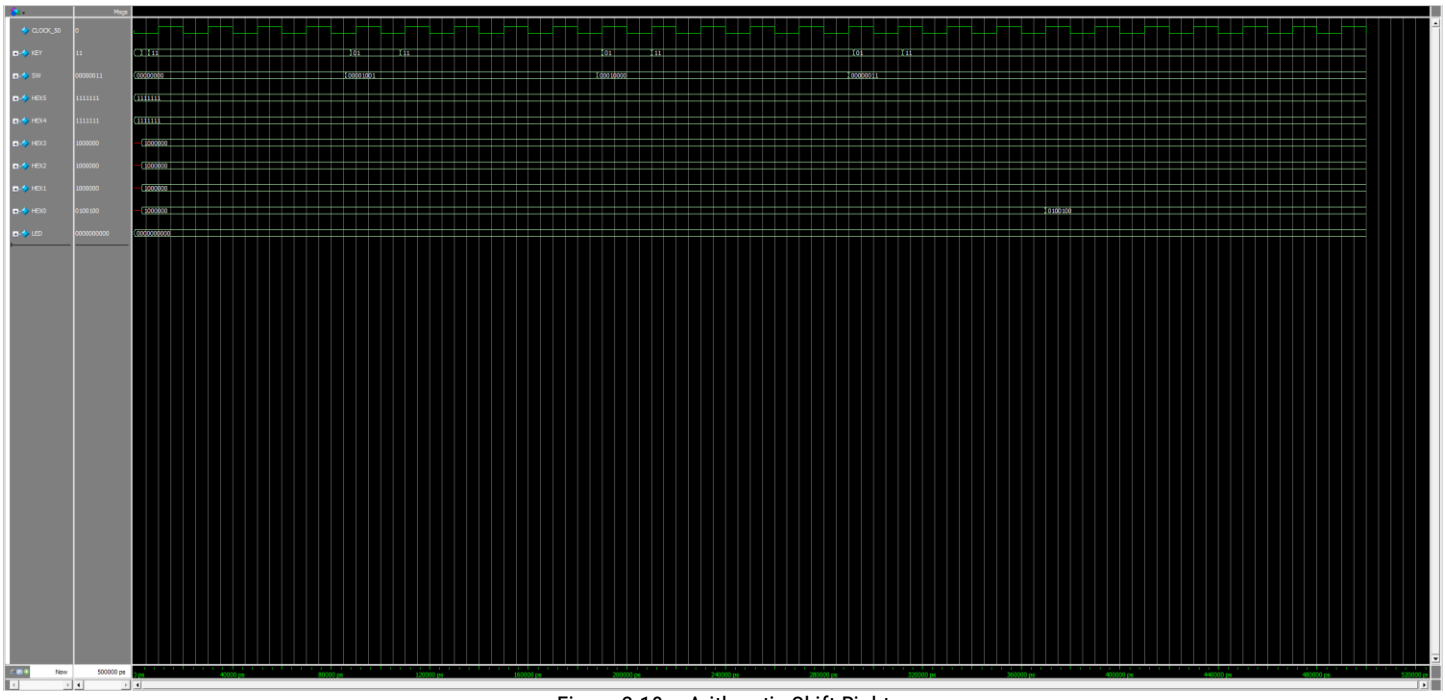

Figure 2-10 – Arithmetic Shift Right

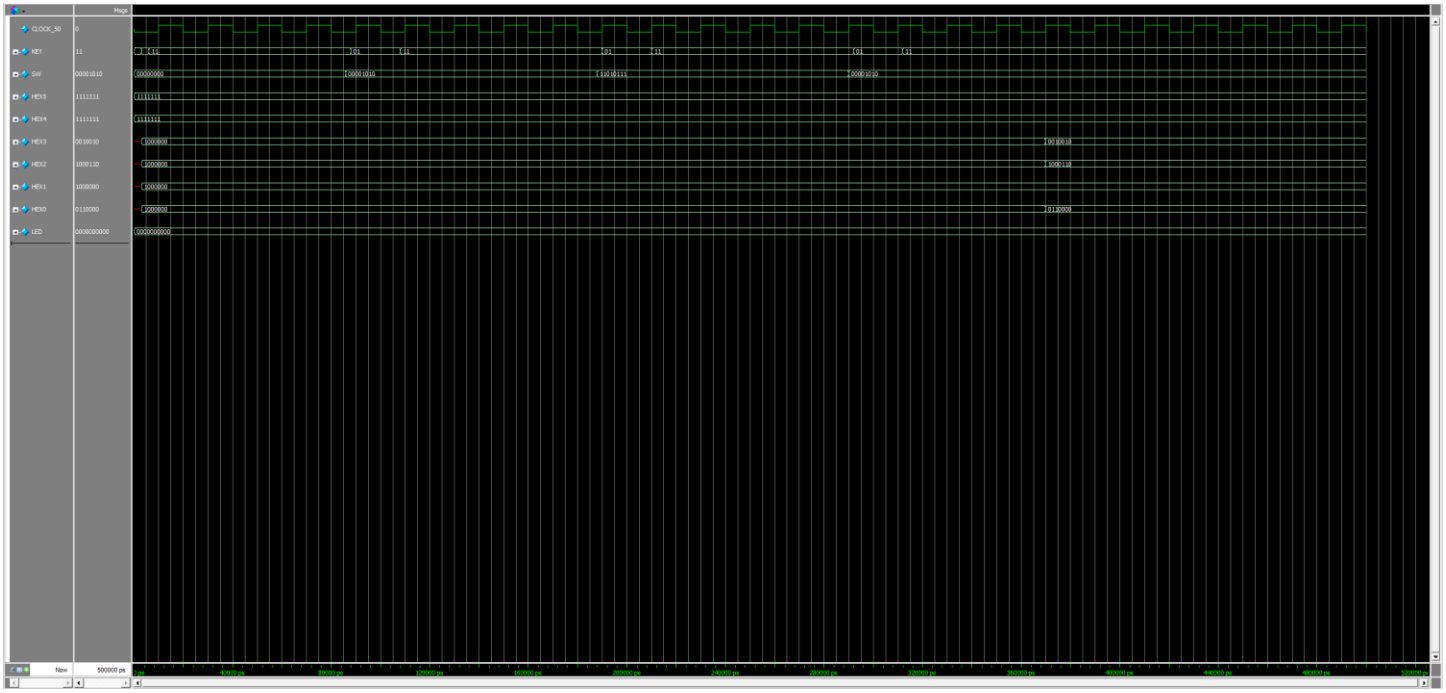

Figure 2-11 – Rotate Left

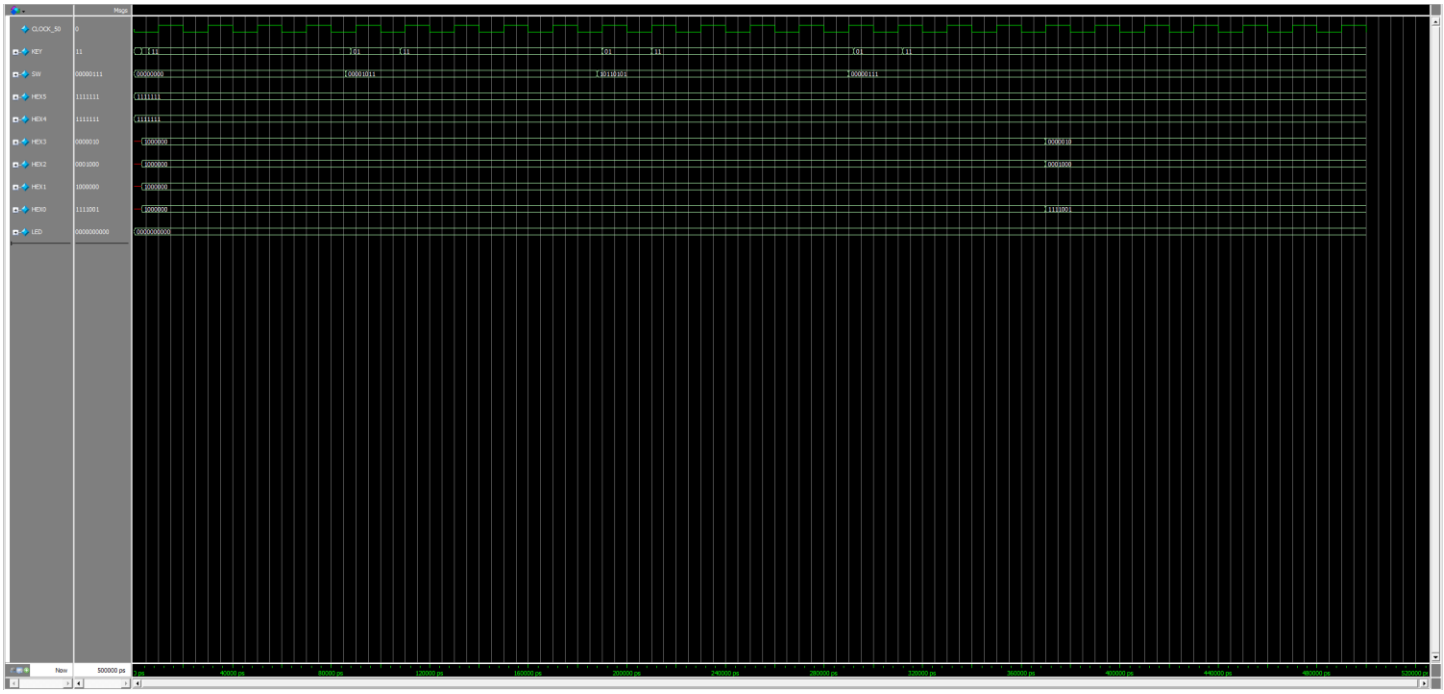

Figure 2-12 – Rotate Right

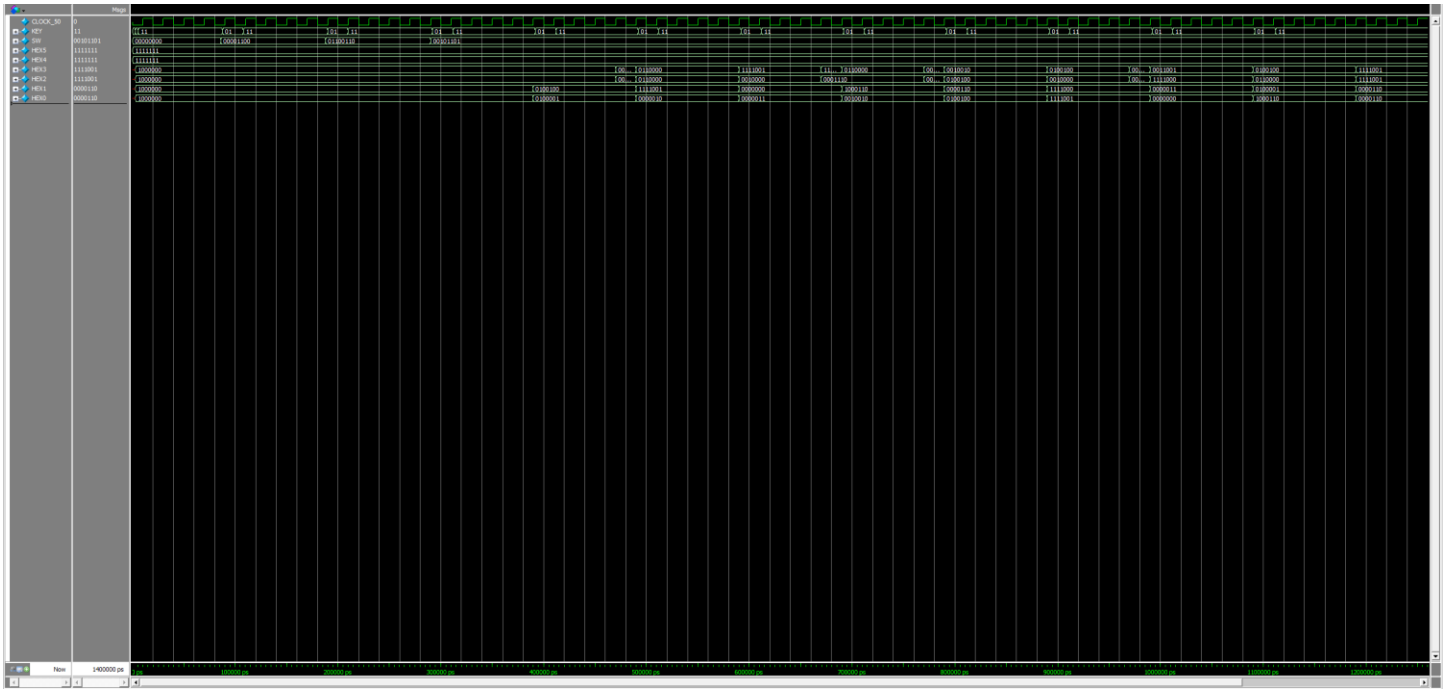

Figure 2-13 – Multiply

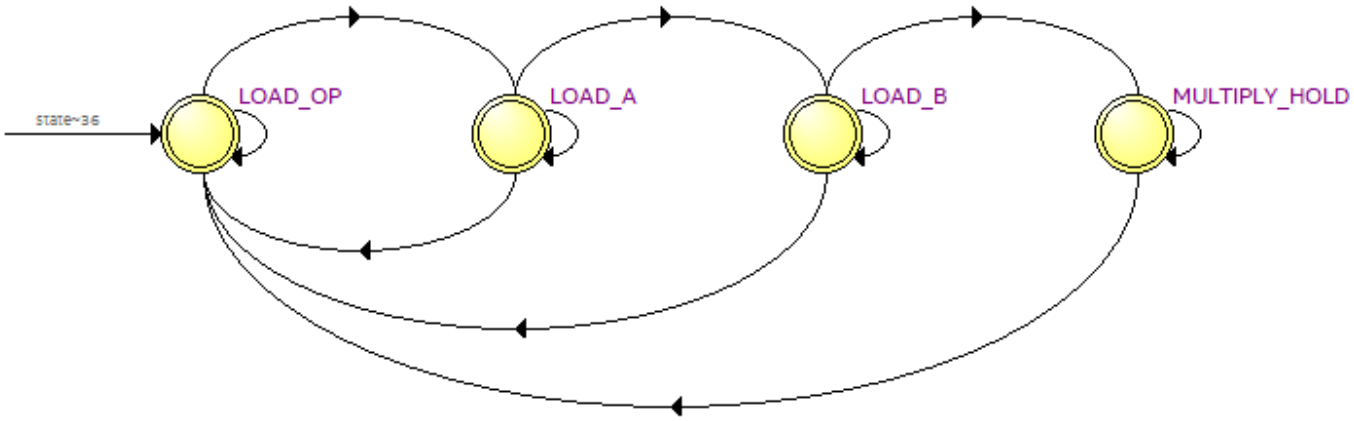

Figure 2-14 – Final State Diagram

Conclusion

Overall, Project 5 was great practice for designing interacting FSMs. Most of the hardware was already built in Homework 7. Modifying the modules wasn't difficult, so the main challenge was building the FSMs around said modules. And a challenge it was.

As I mentioned earlier, there's a consistent timing issue I seem to run into that requires me to add buffers into my state machines when they're actually implemented. On paper, the diagrams work well and function correctly. In both project 4 and project 5 here, one FSM was always one cycle ahead or behind of the other which caused the entire system to fail. Specifically in this project, the control signal output of the FSM would lag behind and cause an ALU register enable to stay on longer than it should've and be overwritten. The same problem happened when coordinating the enable and finish signals to/from the FSM and the multiplier module. There was always a one cycle mismatch that would cause the FSM to skip through its hold state or cause the multiplier to run forever.

I know it's a problem now and I've become pretty good at debugging it now thankfully, but I'd like to catch these problems before I even start writing code. I've dodged bullets on projects 4 and 5 in that I haven't had to rewrite huge sections of my system, but a bad FSM somewhere could easily cause catastrophic problems in larger systems. It's something I'd like to remedy before I take Digital Design 2.

In better news though is that besides that timing issue, writing interacting FSMs is becoming second nature to me, along with writing the Verilog to describe them. My only real experience prior to this class with FSMs was from Microcontroller Interfacing. In that class they're described differently as they're software states instead of hardware states. The biggest hurdle for me to overcome was having outputs change based on the states as opposed to the transitions between states. I'm no expert yet, but as I said, it's starting to become more intuitive to write hardware states. Digital Design 2 will give me plenty of practice if nothing else.