Project 2: Text Scroller

Jacob Boline

October 30, 2018

1 Introduction

In this project I was tasked with creating a design that would scroll text across the 8 digit, 7 segment display. It was required that on power-on the device must load a default message to display and begin scrolling it across the display. In addition, the display must have a mode where the user can program all 8 digits using the 16 switches that were on board. Finally, the device was required to utilize a Block-RAM module for storing the data. To accomplish this project Vivado 2017.2 was used to simulate, synthesize, and implement System Verilog code for a Basys 4 DDR development board. The design files for this project can be found at https://github.com/txjacob/SoC_FPGA

2 Experimental Plan

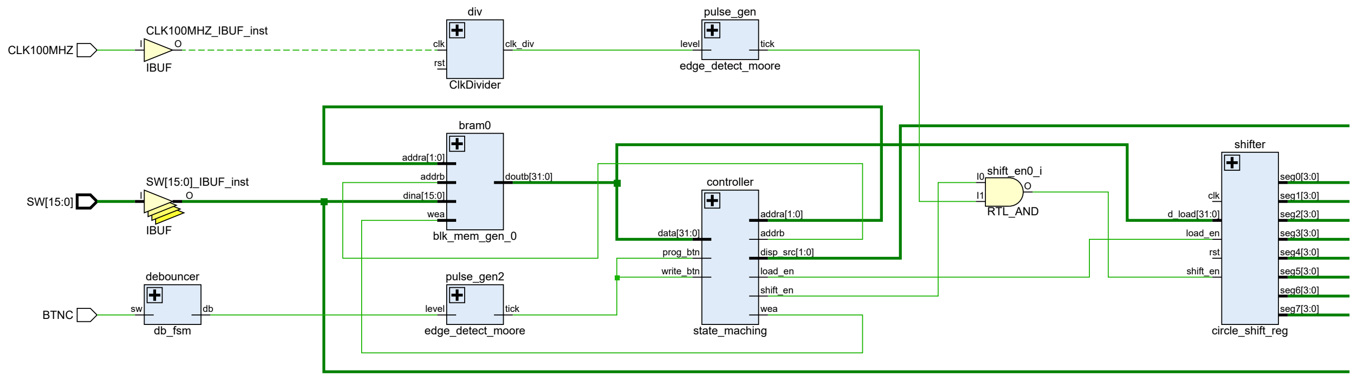

Figure 1: Simplified block diagram of the input and control logic.

Figure 1 shows the input and control logic for the system. The most important module is the controller which is comprised of a state machine that has 8 states: init, load, run, prog0, prog1, inter0, inter1, and load_reg. A diagram for this state machine can be seen in Figure 2.

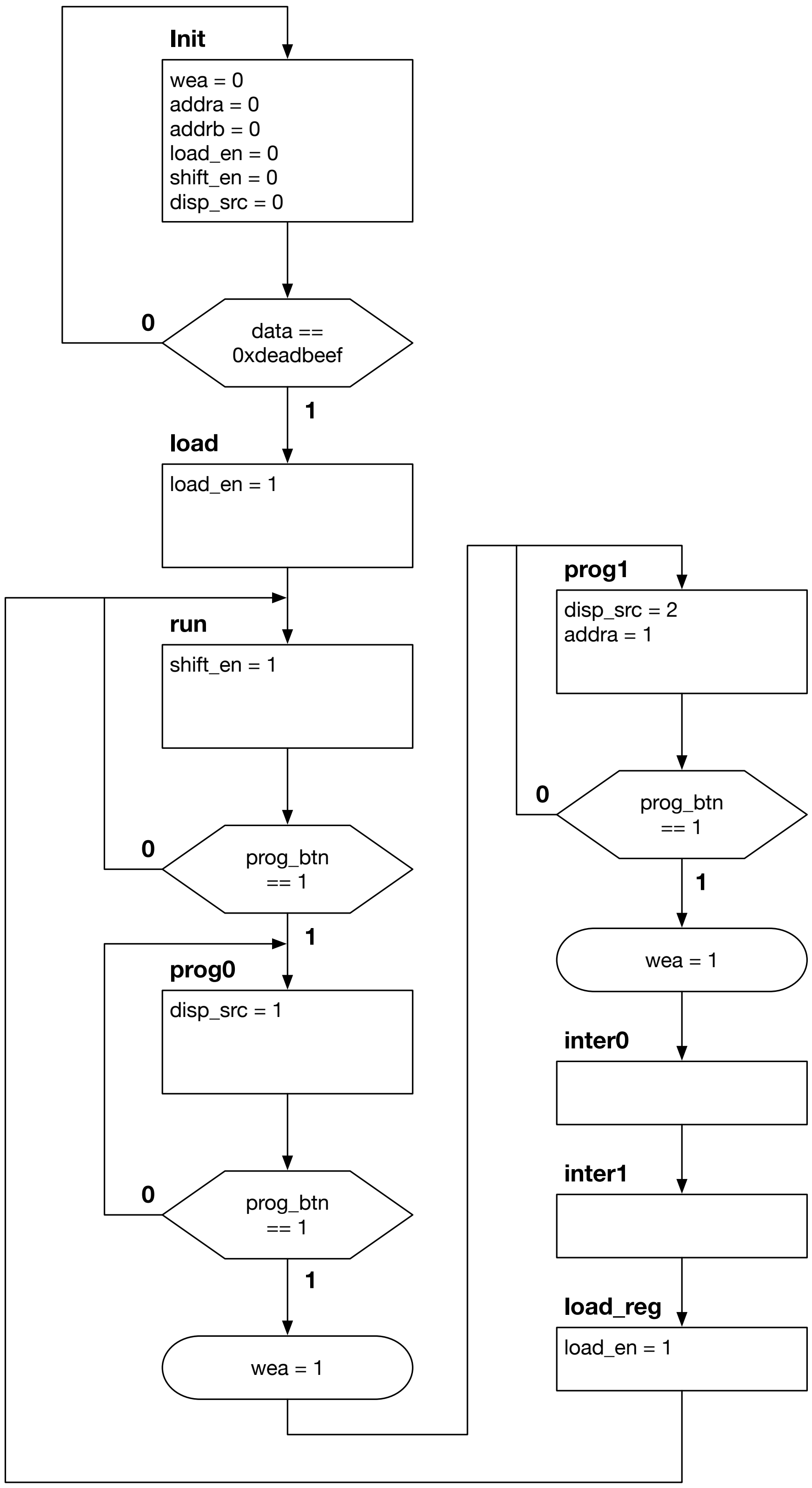

Figure 2: State diagram for controller.

When the FPGA is first programmed the state machine will sit in the init state until the BRAM has properly initialized. Once this has occurred the state machine will load the shift register and move onto the run phase. The programming mode is controlled with the use of a single button. To achieve this functionality I fed the button through a debouncer and into a rising edge detector. This allows a short pulse to be generated when the button is pushed thus only triggering one action from the state machine. When the button is first pushed the programming mode is entered and the bottom 4 digits can be programmed. Once the button is pushed the data from the switches is latched into address 0 of the BRAM, and the top 4 digits can then be programmed. Pushing the button again causes the data from the switches to be latched into address 1 of the BRAM. After the programming state is left, the state machine goes through two delay states. The purpose of these states is to compensate for the 2 cycle latency of the data being programmed into the BRAM. Without them the shift register will try to load the data before it is ready.

The shift register features parallel loading, and shifts data in 4-bit chunks. This was created by building each 4-bit register individually than wiring them up to create an 8 element deep shift register. When wiring up the shift register I realized there would be a noticeable delay if I used the divided clock to load and shift the register. In order to overcome this I instead ran the shift register off of the main 100 MHz clock, and fed the divided clock through an edge detector and into the shift enable pin. Since the clock divider module will always be running I fed the shift_en bit from the state machine through an AND gate with the clock divider to allow the shifting to be completely disabled.

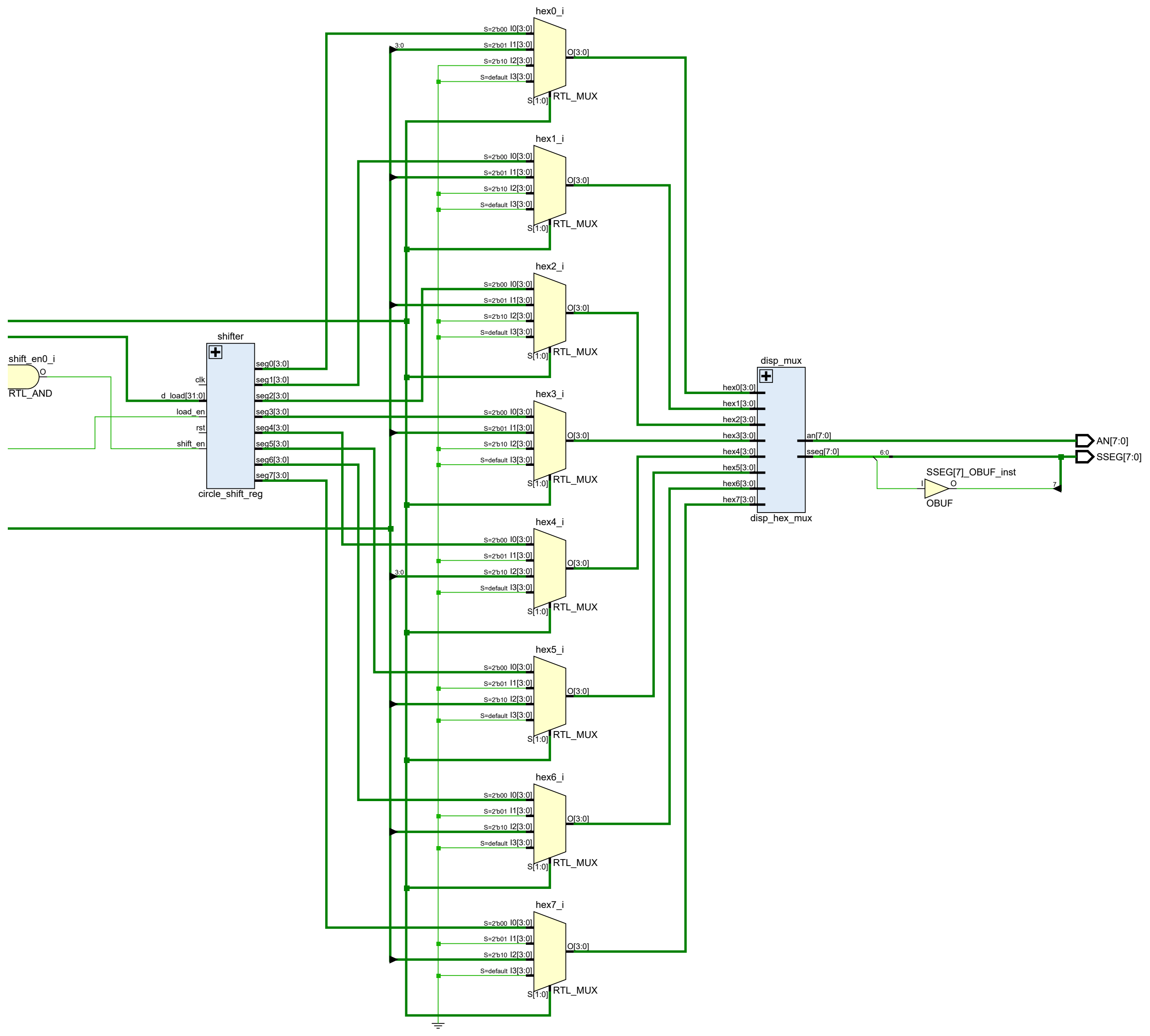

Figure 3: Simplified block diagram of the output logic.

Figure 3 shows the output control logic. The main 2 components are the shift register and the hexadecimal to display module. Connecting each digit of these two components is a mux. This mux is controlled by the state machine and allows the the input of the hexadecimal to display module to be controlled by the state machine. When the address line is 0 all 8 digits are driven with the output from the shift register. When the address is 1 the bottom 4 digits are driven from the 16 switches and the top 4 digits are tied to ground. For an address of 1, the bottom 4 digits are tied to ground, and the top 4 digits are driven from the 16 switches. Finally, all 8 digits are tied to ground for all other addresses. The output of the hexadecimal to display module is hooked directly up to the 7-segment, while the decimal points are tied high to turn them off.

3 Analysis

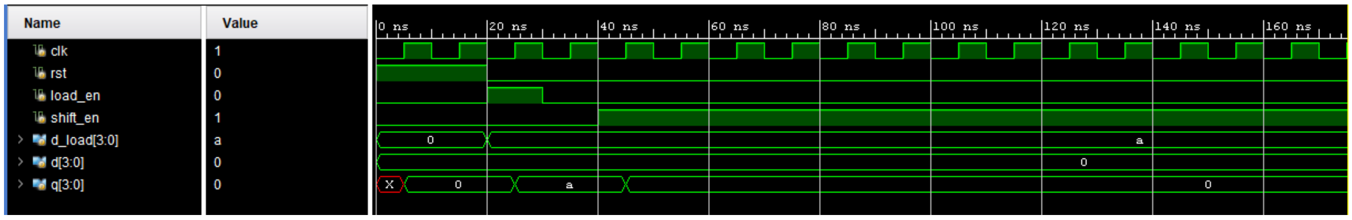

Figure 4: Simulation of a single loadable 4-bit register.

Figure 4 shows the ability of a single 4-bit register to load in a nibble, hold it, then shift it out once shift_en is asserted.

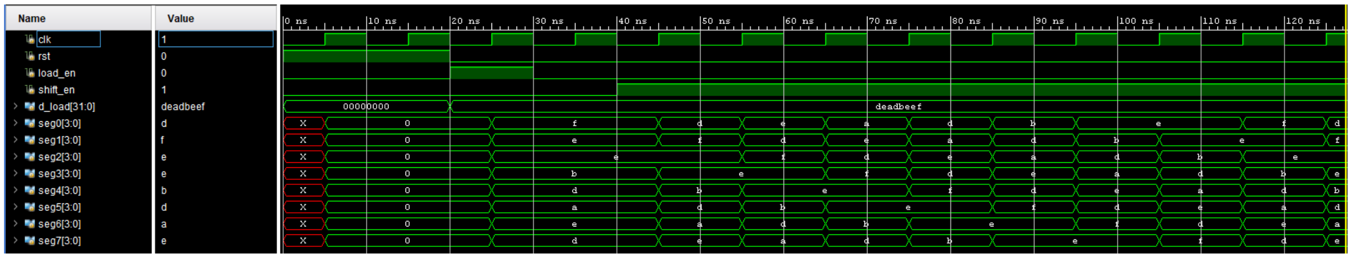

Figure 5: Simulation of a 32-bit parallel load, parallel output shift register.

Figure 5 demonstrates the 32-bit shift register's ability to load in one 32-bit word, hold it, and shift it in 4-bit chunks.

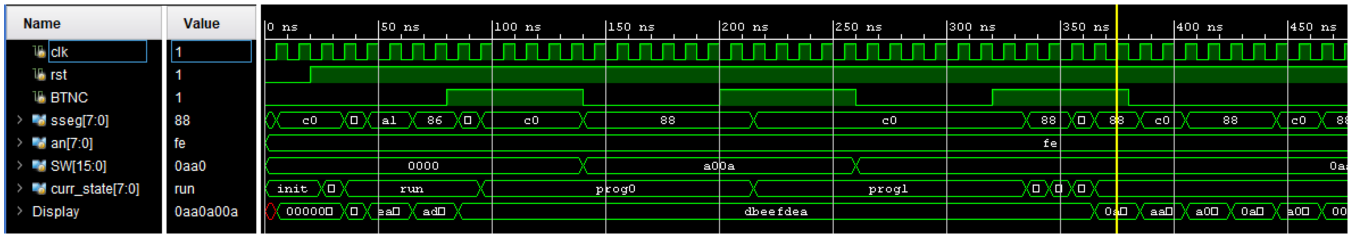

Figure 6: Simulation of the entire system.

Figure 6 demonstrates the full functionality. When the system first comes up it goes into the run state once the BRAM is initialized. Then when BTNC is pressed the prog0 state is entered. After BTNC is pressed again, prog1 state is entered. Finally, after BTNC is pressed again there is a 2 cycle delay, a shift register load, and then the system resumes normal operation. Notice that the long button presses do not trigger multiple state changes.

4 Conclusion

From this report the design process and verification for the system can be seen. The goal of this was to create a system that was simple to put together, and create control logic for. While I believe my design has met these goals I believe there is still room for improvement. My method for waiting for the

BRAM to initialize before loading the shift register is somewhat of a "hack" and causes the system to stop functioning after a systemwide reset. This is due to the BRAM not reinitializing with the set data values from the MMI file.