ECE 440

Project 3: GCD System

Submitted By :
Cory Holt
github.com/cholt150

Contents

1 Introduction 2
2 Design Process 2
3 Design 3
3.1 Input Controller 3
3.2 Output Logic 4
4 Verification 4
4.1 Errors 6
5 Conclusion 6

1 Introduction

This project was a logical conclusion to project 2. We developed a top-level module to process the inputs and outputs of the GCD core module from the previous project. This project had us continue to use SystemVerilog concepts to implement an input controller and output logic.

2 Design Process

I began this design by focusing on the overall structure of the design. From the specified end behavior, I determined that this wrapper module would need 3 main sections: a FSM to handle the input processing, a combinational logic block at the end to handle displaying the output on the on-board LEDs, and a section that will debounce and synchronize the button inputs. Thankfully, the debouncing SV code was given, so I didn't have to worry much about implementing that portion. I began by drawing an (overly simple) block diagram to clarify to myself how I wanted the different processes to interact. Next, I laid out a transition diagram for my FSM

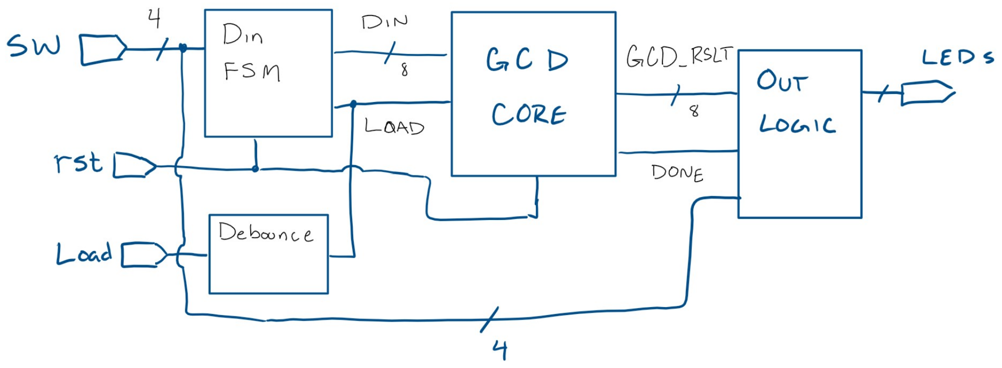


Figure 1: Block Diagram

input controller. This ended up changing several times throughout my process, as I ran into logical issues or mistakes in the assumptions I had made. The final state diagram of my FSM is below.

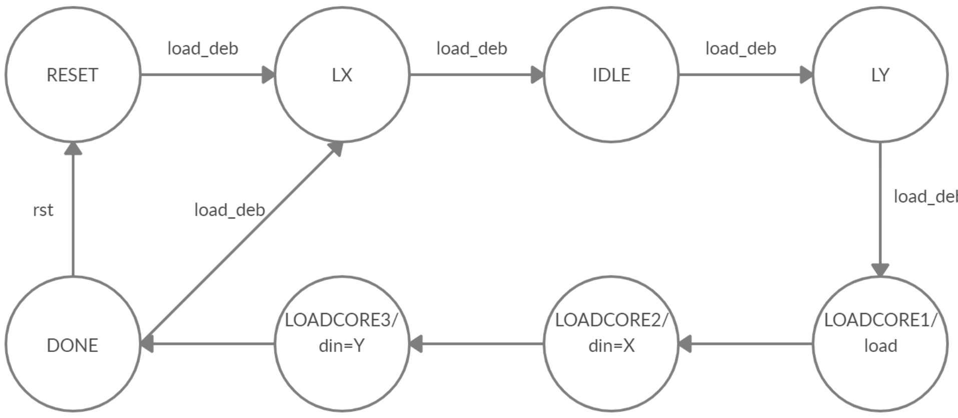


Figure 2: State Diagram

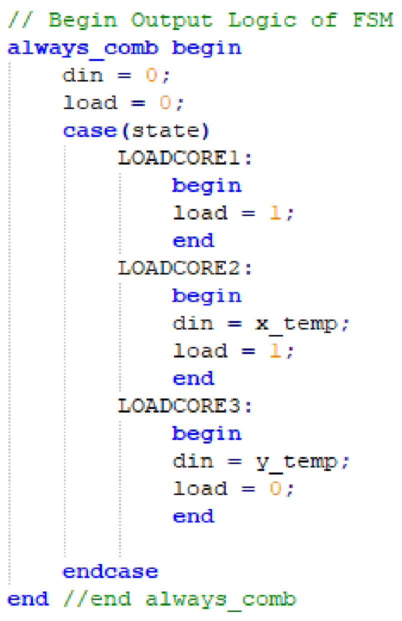
After this initial planning, I began to implement the design in SystemVerilog.

3 Design

I made this project into one file, with instances of the gcd_core and button debouncing modules. It could have easily been broken up into separate files for each portion, but I chose to keep it all as is. There are three main blocks of the system: the input controller, the debounce module, and the output logic. As the code for the debounce module was given, I will discuss the two sections I designed myself.

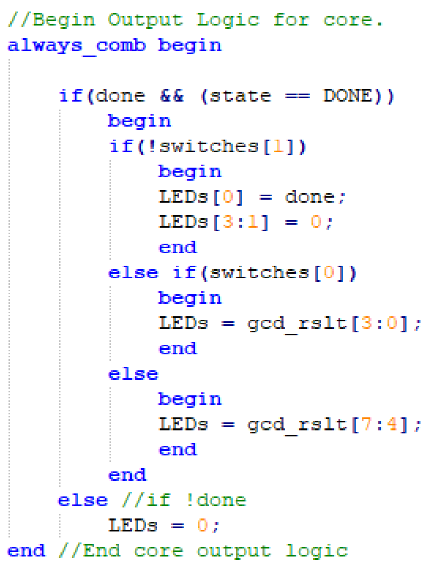
3.1 Input Controller

The goal of the input controller is to accept data from the outside world, in the form of switches, and load it into gcd_core following the design parameters of the core. That means that two 8-bit numbers must be loaded in 4-bit chunks. The controller must remember the values and load them to the core sequentially, following a load signal. To do this, I created a state machine with 8 states. I could have used fewer states, but spelling out each of the steps and clearly working out what each state accomplishes helps me not make mistakes when designing. The state machine is composed of an `always_ff` block and an `always_comb` block. The sequential section handles the next state logic, and the combinational block handles the output logic, which in this case sends the X and Y values to the core after the load signal. The state machine's first four transitions are triggered by the `load_deb` signal, and the rest of the transitions are done at each clock edge. The state machine is a Moore machine, so it has 3 states for the input sequence, and three states for the load sequence. The output logic is modeled by the following:



```
// Begin Output Logic of FSM
always_comb begin
    din = 0;
    load = 0;
    case(state)
        LOADCORE1:
            begin
            load = 1;
            end
        LOADCORE2:
            begin
            din = x_temp;
            load = 1;
            end
        LOADCORE3:
            begin
            din = y_temp;
            load = 0;
            end

    endcase
end //end always_comb
```



```
//Begin Output Logic for core.
always_comb begin

    if(done && (state == DONE))
        begin
        if(!switches[1])
            begin
            LEDs[0] = done;
            LEDs[3:1] = 0;
            end
        else if(switches[0])
            begin
            LEDs = gcd_rslt[3:0];
            end
        else
            begin
            LEDs = gcd_rslt[7:4];
            end
        end
    else //if !done
        LEDs = 0;
end //End core output logic
```

3.2 Output Logic

The output logic is simply a combinational block. It boils down to two 4-bit 2:1 multiplexers. The first mux determines whether or not to display the done signal on `LEDs[0]` and the second mux determines which nibble of the result to display on the LEDs. This logic is implemented by the above right code.

4 Verification

The first verification I performed was running behavioral simulations. I tested several values to verify that the core was reading the values properly. I also tested large values to ensure that all relevant bits were being read. The final values I tested were 254 and 127. This resulted in a GCD of 217, which allowed me to verify that the output logic was behaving as expected, and displaying all bits of the result. The simulation results are shown below. I then ran synthesis and verified that there were no critical errors in synthesis. I also ran one post-synthesis simulation to verify that the results matched my behavioral sims. Finally, I created the bitfile and programmed the board so that I could physically verify the behavior of the wrapper. The behavior of the programmed board matched my simulations, and I was able to find and verify the GCD of various values. Satisfied that it worked well enough, I wrapped up the project.

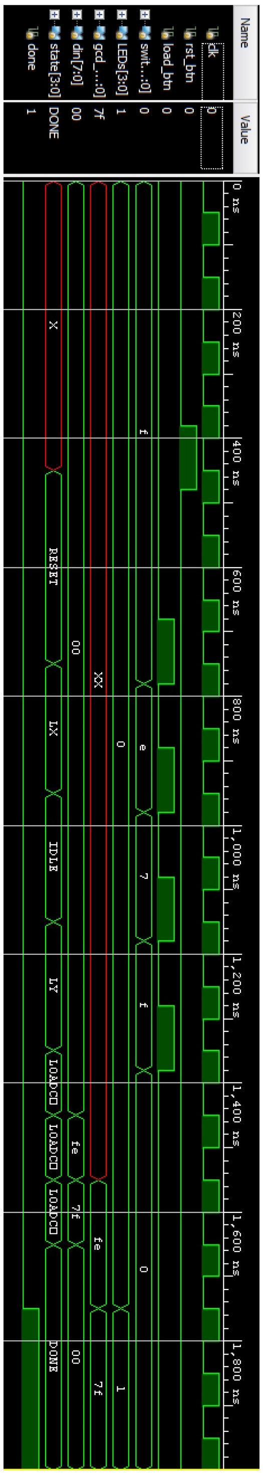


Figure 3: Behavioral Sim

4.1 Errors

While there were no final problems with my project, I ran into many issues when implementing it. Firstly, I designed my loading FSM with a different loading sequence in mind. I mis-remembered the specifications of the core, and had my design centered around sending two load signals for each value. Fixing this required me to add more states to the machine. Another issue I had with the FSM was timing related, where I neglected to implement the proper timing for the load signal, so the load signal was sent to the core concurrently with the X value. This led to the Y value never loading, as the process was off a clock cycle. Adding another state corrected this issue.

Another problem I had was related to the constraint file. When running implementation, Vivado returned an error about the clock signal. As it turned out, I had incorrectly uncommented lines in the constraint file, and two of the required lines were essentially missing. Simply uncommenting the lines fixed the problem and the project implemented without errors.

Finally, although this is less of an error than unwanted behavior, I added a condition to the LED output logic that checks the current state. Previously, the LEDs held their value while entering the new values, which was kind of confusing. So I added a condition for the "DONE" state of the input FSM, so results are only shown to the LEDs when the machine is not in a loading process.

5 Conclusion

This was a challenging project, as I had to combine most of what we have worked on so far into a single file. Working on this project taught me lots about the software tools and the styles and practices I prefer when implementing a design. I learned how to resolve common problems and how to identify the sources of warnings and errors. I liked that we were able to actually test our design on the board and work our way up the process to that point. Overall, this was a helpful project for learning more about the SystemVerilog design process.