ECE4700J Computer Architecture

Summer 2022

Lab #2 Some Verilog Debugging and Testing

Due: 11:59pm Jun. 3rd, 2022 (Beijing Time)

Logistics

- This lab is an individual exercise.
- All the design code should be in SystemVerilog.
- This lab must be checked off by a TA before end of lab on Friday, Jun. 3rd, 2022 (Beijing Time). It is highly recommended you complete the lab before the following week's lab release.
- All code and reports (if available) MUST be submitted to the assignment of Canvas.
- Internet usage is allowed and encouraged.
- No late submission is allowed for this lab.

Overview

In this lab, you will study how to debug Verilog codes and how to write good testbenches.

- Study different kinds of bugs in hardware design.
- Correctly debug the given buggy designs.
- Be able to implement testbenches which both have good coverage and short execution time.

Introduction

This lab is on debugging and testing. We will cover several ways of debugging errors in the lab. An error inside the SystemVerilog files can be either a syntax or a logical error. You will need find the errors in the finite state machine in Part A and the small design in Part B.

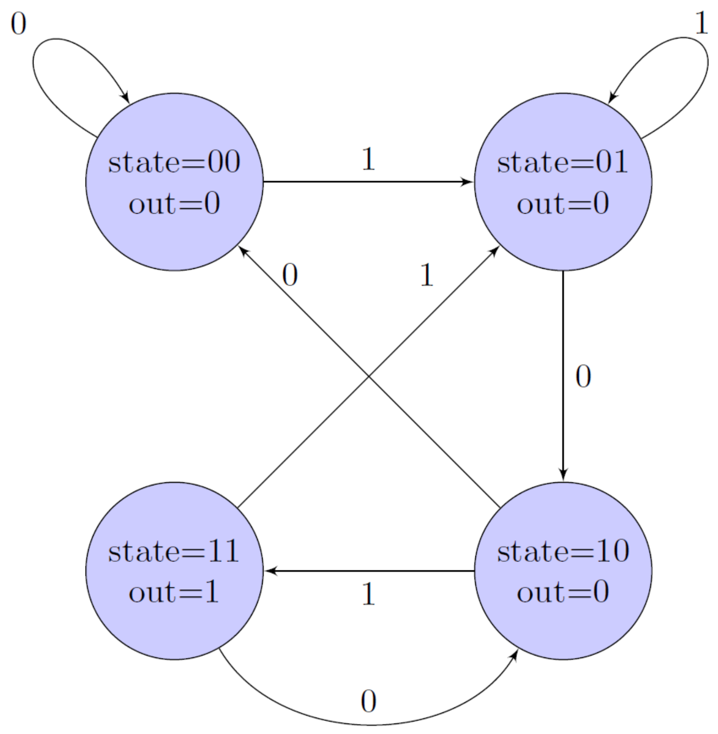
Initially you are given a .tar file with all the files needed to run and test Finite State Machine (FSM) A. You should extract this tar file to a folder. This folder only have the code for Part A, since in Part B we only require you to tell your answer during live demo.

Assignments

I. Debugging of Verilog Design (Mandatory)

a) Part A: Syntax and Logical Bugs

Hardware debuggers will meet bugs very often. Some bugs are due to bad syntax, wrong logical expressions, or bad coding styles. These bugs are relatively simple to be found. Your assignment part A is to fix the errors in an FSM as shown below.



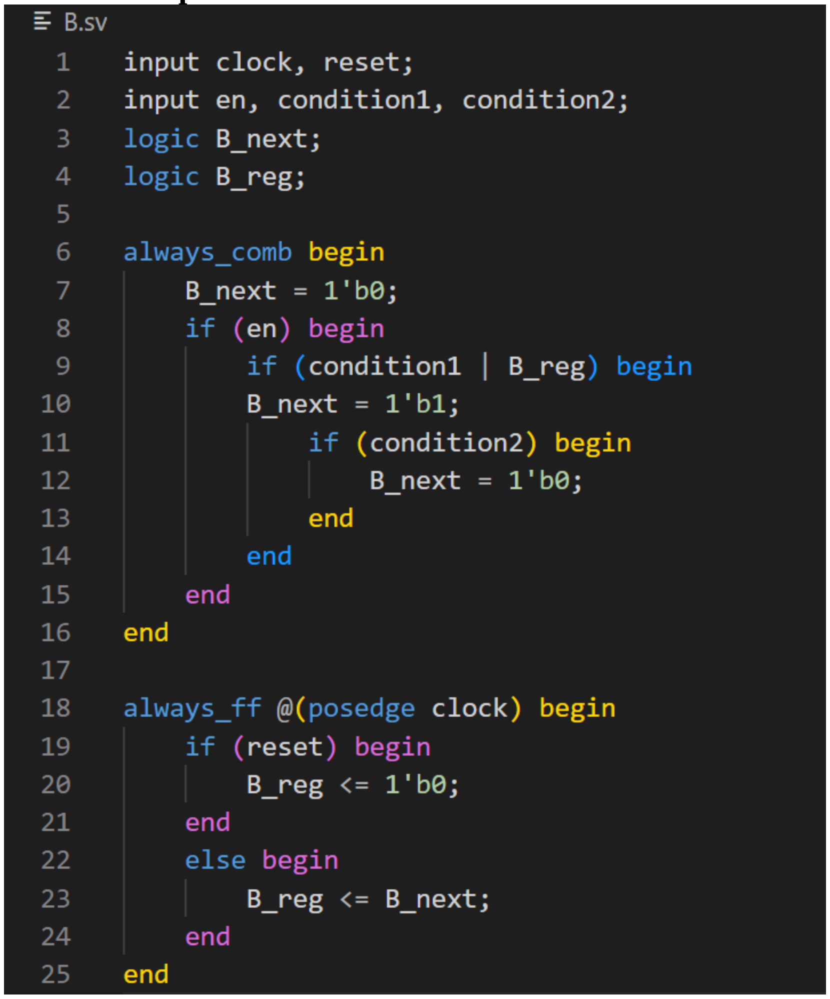
The reset signal should send the state machine back to state=00. The transition signal for this state machine is labeled in. The buggy design can be found in fsm_ab.sv. We also provided you a testbench, test_ab.sv. Read and understand how this testbench is used.

You need to fix the bug inside the design(fsm_ab.sv), and also make it to be in good style. (Vivado is an advanced tool, it sometimes can automatically fix your syntax problems, like using blocking assignments in sequential logic). **You need to pass post-synthesis simulation of the design by using the given testbench, i.e., your simulation must end normally after passing all tests.** (Hint: Your Vivado's default simulation time 1000ns may not be enough for your testbench to finish. You need to extend Vivado's simulation time in "Settings".)

b) Part B: More Complex Bugs

Unfortunately, sometimes the cause of bugs is not trivial. Sometimes you may forget one condition, or you may schedule something wrongly… These bugs are hard to detect. Your assignment part B is to help find a bug in a small design:

- **Specification**: ***B_reg*** should be initialized as 0. When ***en***&***condition1*** is true, ***B_reg*** should be set as 1 (start from the next cycle), and value 1 will be kept until sometime ***en***&***condition2*** is true, then ***B_reg***'s value should be set back to 0 (start from the next cycle). It's guaranteed that condition1 and condition2 cannot both be true at the same time. During the time when the value of ***B_reg*** is 1, the input ***en*** signal can be set as 0, but this should not affect the value of ***B_reg*** to be 1.
- **Code Implementation**:



B.sv

```
input clock, reset;
input en, condition1, condition2;
logic B_next;
logic B_reg;

always_comb begin
    B_next = 1'b0;
    if (en) begin
        if (condition1 | B_reg) begin
        B_next = 1'b1;
            if (condition2) begin
                B_next = 1'b0;
            end
        end
    end
end

always_ff @(posedge clock) begin
    if (reset) begin
        B_reg <= 1'b0;
    end
    else begin
        B_reg <= B_next;
    end
end
```

However, the code is buggy. **Please think carefully and prepare your answers to the following questions for your live demo**:

- Why this code implementation is wrong? Give your explanation.
- We only need to change one line to fix the bug. How to do it?

This bug is actually from a real bug in the development of a famous open-source AXI-Stream Library, you can see the details here: https://github.com/efeslab/hardware-bugbase/tree/bugs/d12-failure-to-update-frame-fifo. However, it's just a very simple one. There are many different kinds of bugs in hardware design.

If you are interested in debugging hardware designs, Jiacheng et al[1] have conducted a hardware bug study, they basically divides all the hardware bugs into 3 main categories: Data Mis-access Bugs, Communication Bugs and Semantic Bugs. Their work (Debugging in the brave new world of reconfigurable hardware | Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems) also introduces some new tools that can help people debug their hardware design. If you are interested, you can take a look.

II. Testbench Writing (Optional)

In this optional assignment, you will be designing some testbenches. **You will be given an up to 10% bonus to the total score of Lab 2** if you are able to correctly implement the optional assignment. This will be counted towards your final grade.

1. Writing a Robust Testbench

Some modules are simple enough that you can write a testbench with complete coverage (The more conditions you have tested, the higher coverage your testbench has. Complete coverage means that your testbench have tested all the possible input conditions.). Often times, you will have to strike a balance between a thorough testbench and the amount of time required to generate the testbench, or for the testbench to run. We have supplied you with a slightly obfuscated full-adder (fa1.sv) and a stub testbench (fa1_test.sv). You need to write fa1_test.sv such that it achieves complete coverage of the full-adder module. Once you have done this, it should be a quick fix to identify and fix the bug in fa1.sv.

2. Using Arrays of Modules

Using the 1-bit adder from Part A, build a 64-bit adder as shown in the lab lecture slides. The file named fa64.sv contains a module stub full adder 64bit. Make the module into a complete 64 bit adder.(This should take no more than 5 minutes, but if it does, please ask around for help and discuss with your neighbors.)

3. Writing a Harder Testbench

We have provided you with a slightly more complete fa64_test.sv. As a first step, read through the code we provided and make sure you understand it.

Try running the testbench. What’s happening? Is 100000 ns enough for you to finish your test? Why?

Since our testbench won’t work as written, what kind of tests could you add to achieve good, but imperfect coverage? Modify fa64_test.v to be a useful testbench and make sure your fa64.v implementation passes(post-synthesis test). Specifically, you need to add some specialized test cases. What specific test cases are useful and/or a good idea to validate your adder? Be prepared to defend your choices. Our compare correct sum task is incomplete. You need to identify what it’s missing and fix it.

References:

1. Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Haoyang Zhang, Andrew Quinn, Baris Kasikci, "Debugging in the Brave New World of Reconfigurable Hardware", *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.(ASPLOS '22)
2. Umich EECS470 WN 2021 Lab2
3. Umich EECS470 WN 2021 Lab3

Acknowledgement:

- Jon Beaumont (University of Michigan)

Deliverables:

- Live Demo of Mandatory Assignment:
    - Show TA with the waveform of your post-synthesis timing simulation of your part A, and give answers of part B questions.
    - If you implemented the optional assignment, you may need to show TA your design.
- Individual Deliverables:
    - Submit design files of the Mandatory Assignment: fsm_ab.sv (contrasted into a zip/tar file) via canvas assignment.
    - If you choose to finish the optional assignment, submit your design files and testbench files (fa1.sv, fa1_test.sv, fa64.sv, fa64_test.sv) via canvas.

Grading Policy

Live Demo – 70%
Canvas Files Submission and Correctness – 30%