Stony Brook University

Project 1: SystemVerilog

Issued: 9/14/15, Due: 9/28/15 4:00PM

Introduction

The objectives of this assignment are (1) to gain experience creating a small design in SystemVerilog, testing it, and evaluating it through synthesis and (2) to answer questions related to course topics. You will turn in:

- your documented code including testbench
- clearly labeled synthesis reports
- a short report answering all questions and including the information requested below

I will run additional simulations on the code you turn in, so it is very important to:

1. Make sure the names and behavior of all signals matches the specification in this handout
2. Carefully label and document your code
3. Create separate subdirectories for your code for parts 2 and 3
4. Include a README file in each directory giving a description of each file, and the exact commands you are using to run simulation and (where appropriate) synthesis.

Your project will be evaluated on correctness ***and efficiency*** of your designs, the thoroughness of your testbench, and your report/answers to questions.

You may work alone or with one partner on this project. **You may not share code with others (except your partner). All submissions will be run through an automatic code comparison tool.**

If you have general questions about the project please post them Piazza.

Part 1: Questions

Answer the following questions in your report.

1. Consider the system shown in Figure 1. Assume each flip-flop has a 2ns propagation delay, a 2 ns setup time, and a 4 ns hold time. What is the shortest possible clock period? What is the fastest possible clock frequency? Explain your answers clearly.

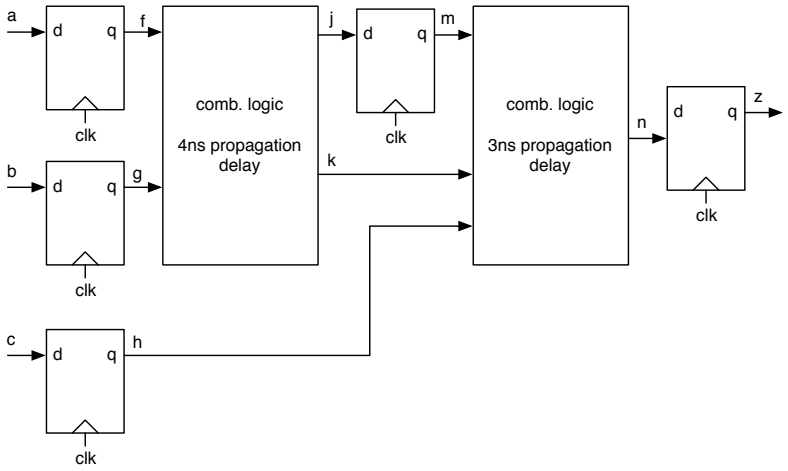


Figure 1. (Problem 1)

2. Explain why the following SystemVerilog module will not synthesize as a combinational circuit. You may use a synthesis tool to understand this, but don't just paste a synthesis message into your report; make sure you understand and explain what the problem is. Show how you could change the description to make it a synthesizable combinational circuit.

```
module mod1(sel, f0, f1, f2, f3, a);
      input [1:0] sel;
      input a;
      output logic f0, f1, f2, f3;

      always_comb begin
            case(sel)
                  2'b00: f0 = a;
                  2'b01: f1 = a;
                  2'b10: f2 = a;
                  2'b11: f3 = a;
            endcase
      end
endmodule
```

3. Complete the tasks from problem 2 again, but for the following SystemVerilog module:

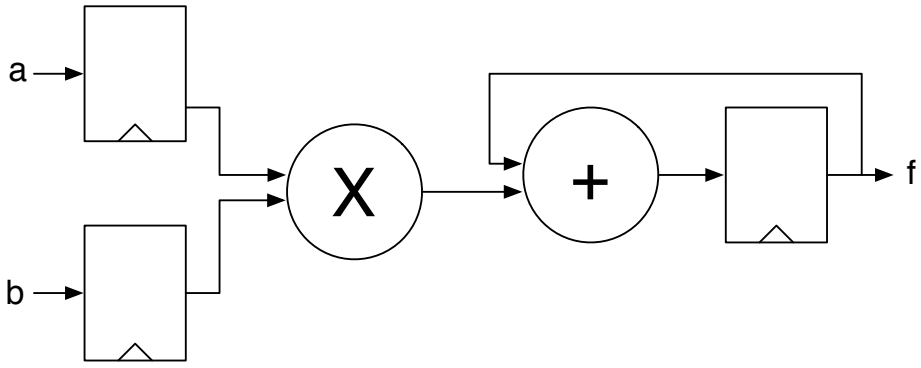
```
module mod2(a, b, c, d, e);
   input a, c;
   output logic b, d, e;
   always_comb begin
      if (c == 1) begin
         e = a;
         b = c;
      end
      else begin
         e = 0;
         b = a;
      end
   end
   always_comb begin
      d  = a | c;
      b  = e ^ a;
   end
endmodule
```

Part 2: Basic Multiply and Accumulate

Multiply and accumulate (MAC) is a basic operation used frequently in many different types of computations. MAC is defined as

$$f = f + a*b,$$

where a and b are inputs to the system, and f is the output. Obviously, the system will require feedback because f depends on the previous version of f (this is "accumulation"). You will build the following system:



Clock and reset signals are omitted in this figure, but will be needed. Note that we also have registers on the inputs a and b. So, in one clock cycle, a and b will flow out of their registers, through the multiplier, through the adder (where they are added with the previous value of f), and into the output register.

Here, assume that *a* and *b* are each 8 bit signed values. The output of the multiplier is a 16 bit signed value, and *f* is a 16 bit signed value. Assume all registers have synchronous reset—they only reset when the "reset" signal is equal to one on a positive clock edge. Assume the reset is positive-asserted (that is, when `reset==1`, the registers reset to 0).

Overflow can occur when the result of an arithmetic operation cannot be represented using the allotted number of bits. For example, you can tell that an adder has overflowed if:

- you add two positive values and get a negative answer, or
- you add two negative values and get a positive answer.

Based on our specified bit widths, our system may over or under-flow in the adder. It is possible to build a piece of logic that looks at the adder's inputs and outputs, and checks for overflow. We will not require this for Project 1, but we may add this in the future.

Use the following module name, port names, and port declarations:

```
module part2_mac(clk, reset, a, b, f);
    input clk, reset;
    input signed [7:0] a, b;
    output logic signed [15:0] f;
```

It is very important that your module matches this input/output specification exactly, or it will fail all of the tests our additional testbench performs.

Your tasks for Part 2 are:

1. Write a module in SystemVerilog that contains this multiply and accumulate system.

2. Write a testbench that will test your module well. Make sure you use enough comments so that during grading, I can run your testbench and tell from its output if your design worked as desired. In other words, make sure your testbench includes comments as to its expected output.

3. Use Synopsys DesignCompiler to synthesize your design. (Use the same scripts from project 0. Don't forget to configure the script at the top with your top module name, clock frequency, and so on). Your goals here are to find the maximum possible clock frequency, and to evaluate the area, power, and critical path location for a number of different frequencies. Make sure you understand how the area and power change as frequency changes.

   It is very important that you correct any synthesis problems reported by DesignCompiler. If you have errors, the tool's output will not be correct. You also must be certain to fix any inferred latches from your design.

   One common synthesis warning that you can safely ignore is
   `Warning: ./proj1.sv:182: unsigned to signed assignment occurs. (VER-318)`

4. In your report, describe/answer the following:

    a) How your testbench works, and why you think it is a sufficient way to test the design.

    b) Most likely, you picked a fairly straightforward way of testing your design. Do you have any ideas of how you could have designed a more robust testbench?

    c) The area, power, and critical path locations you determined for different clock frequencies. Make sure you include units (e.g., um^2). Explain why you chose these frequencies. Make sure you found the maximum reachable frequency.

    d) Show the relationships you found between clock frequency and both area and power. Explain the trends you observed and why they occur. (Graphs would be a good way to show this.)

    e) Write an expression for the *energy per operation* of your system. In other words, how much energy does the system require for every multiply-and-accumulate operation?

    f) Does the energy-per-operation change significantly if you change the clock frequency? Why or why not?

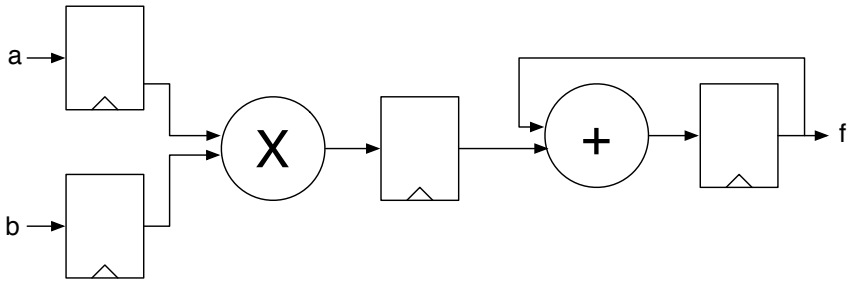
    g) The directions above told you to include reset signals on the registers. Is it necessary for you to do so for the system to work correctly? For all registers? Explain.

Part 3: Pipelining

For part 3, make a new copy of your design from Part 2. Change the module name to `part3_mac` (but otherwise use the same module specification).

Your task for Part 3 is to increase the maximum clock frequency of your MAC unit through pipelining. When pipelining the system, you will break the computation into one or more extra stages by inserting registers into the datapath.

1. The most obvious solution involves adding one extra register between the multiplier and the adder. First, just try inserting this one register:



How will this extra register change the behavior of the output of your system? Do you need to change your testbench? Make sure you test your pipelined design. Then, synthesize it, and note the area, power, frequency, and the critical path.

2. There are other ways to increase the clock frequency through pipelining. For example, the multiplier itself could be pipelined by inserting a register stage *inside* of it. Unfortunately, since we are using the * operator to give us a multiplier, there is no easy way to insert an extra register inside. However, the Synopsys tools provide us with a way to *instantiate* a pipelined multiplier.

   Comment out the lines that perform the multiplication in your design, and instead replace them with the following instantiation of a multiplier.

```
DW02_mult_S_stage #(8, 8) multinstance(input1, input2, 1'b1,
     clk, output);
```

   In this code, replace the `S` in `mult_S_stage` with the number of pipeline stages you want inside of the multiplier (from 2 to 6). Replace `input1`, `input2`, and `output` with the name of your multiplier's input and output signals.

   Try several values of for `S`. (Also keep the register *between* the multiplier and area.) For each, synthesize your design, and find the maximum reachable frequency (and corresponding area). How high of a frequency can you reach?

   Note that once you use the DW02 instantiation for your multiplier, you will not be able to simulate your design in ModelSim. (In order to do so, you will need a *simulation model* for the DW02_mult_S_stage modules.) However you will still be able to synthesize it with DesignCompiler. For this step only, skip the simulation.

3. In your report, make a table to summarize the different designs you tried. For each, explain what you did, and give the maximum frequency and the corresponding area and power. Compare the energy-per-operation of these designs to what you calculated in Part 2.

4. Which is the *best design*? Justify how you chose the "best."

5. It would also be possible to add pipelining to the *adder*. If you did so, would this create any other problems or difficulties?

Code and Report Submission

1. Code

You will turn in a single **.zip, .tar, or .tgz** file to Blackboard. ***Do not use a different archive format (including .rar).*** This compressed file should hold all of the files from your project. Make sure your testbench contains comments that include the expected output. I will be testing your designs using my testbenches, so it is very important that you stick to the specification closely.

Do not turn in things like ModelSim "work" directories or gate-level Verilog produced by synthesis.

2. Report

Your report should include the information requested above. You may include your report in the electronic hand-in with your code (as a PDF file only), or you may turn it in as a hard copy in class.

3. Electronic Hand-in Process

To hand in your code, go to Blackboard –> Assignments -> Project 1. There you can upload your .zip, .tar, or .tgz file. You only need to hand in once per group, but make sure both partners' names are clear in your code and report.

If you want to copy files from the lab computer to your personal computer, please see the section on SCP in the FAQ of the tutorial from Project 0. (FAQ #10 on the last page.)

To create a .tgz file in Linux, first assemble a hand-in directory with copies of all of your code, etc. For this example, let's assume that directory is called `handin`. Now, assuming you are one directory above `handin`, type the following:

```
tar cvzf myhandin.tgz handin/
```

This will create a gzipped-tar file (`.tgz`) that contains the entire `handin/` directory (including all of its contents).

You can test that it worked properly by copying the .tgz file you created to another directory, and typing:

```
tar xvzf myhandin.tgz
```

This will extract the file into the directory you are currently in. If you have any problems with this or anything else, please post them on Piazza.