Stony Brook University

Project 1: Multiply and Accumulate Hardware

Issued: 9/18/17, Due: 10/2/17 4:00PM

Introduction

The objectives of this assignment are (1) to gain experience creating two small designs in SystemVerilog, testing them, and evaluating them 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 would like a private Git repository for you and your partner to use for sharing code, please contact me and I will make an account for you on my server.

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

Partner

If you are choosing to work with a partner, by Wednesday 9/20 at 11:59pm you must:

- Send an email to peter.milder@stonybrook.edu with the subject "ESE 507 Project 1 Partner Signup"
- Send the email from your @stonybrook.edu email address
- In the body of the email, write both your name and your partner's name
- CC your partner on the email (using your partner's @stonybrook.edu email address)

After this, you are committed with working with this partner for project 1. (If you want to change for later projects, you may.)

Part 1: Questions

Answer the following questions in your report.

1. Consider the system shown in Figure 1. Assume each flip-flop has a 2 ns propagation delay, a 3 ns setup time, and a 1 ns hold time. What is the shortest possible clock period? What is the fastest possible clock frequency? Explain your answers clearly.

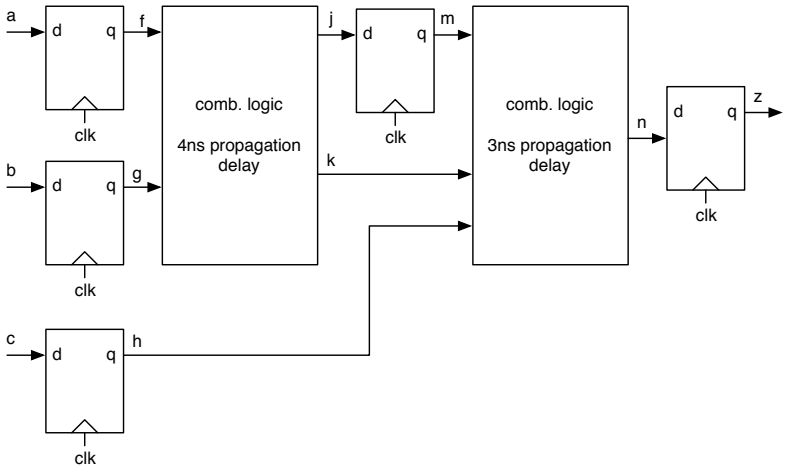


Figure 1. (Problem 1)

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 a small set of changes you could make that would make this a synthesizable combinational circuit (with some change in functionality).

```
module mod1(sel, g0, g1, g2, g3, a);
      input [1:0] sel;
      input a;
      output logic g0, g1, g2, g3;

      always_comb begin
            case(sel)
                  2'b00: g0 = a;
                  2'b01: g1 = a;
                  2'b10: g2 = a;
                  2'b11: g3 = a;
            endcase
      end
endmodule
```

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

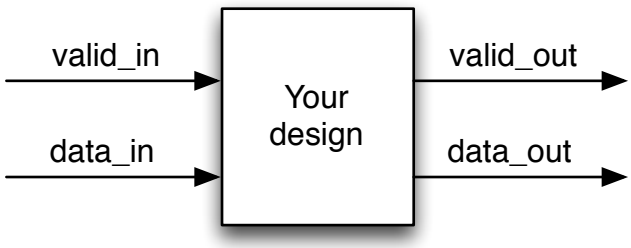
```
module mod2(a, b, c, d, e);
   input a, c;
   output logic b, d, e;
   always_comb begin
      if (c == 1) begin
         e = a;
         b = c;
      end
      else begin
         e = 0;
         b = a;
      end
   end
   always_comb begin
      d  = a | c;
      b  = e ^ a;
   end
endmodule
```

Part 2: Basic Multiply and Accumulate

The goal of this section is to construct a multiply-and-accumulate unit (abbreviated MAC for "**M**ultiply and **AC**cumulate"). Later projects will build on this, using your MAC as a basis for larger systems.

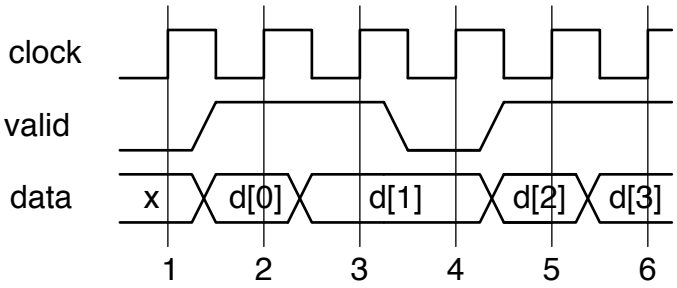
Input/Output Signaling Protocol

Your system will use a simple synchronous protocol to transfer data. (In later projects, this will grow more complex.)



Your system takes one or more data input signals (in this project, it will be two) and a valid_in signal. On a positive clock edge, if valid_in is asserted, then there is valid data to take on the input port. However, if valid_in is 0 on a positive clock edge, there is no input data and your system must halt until input becomes available.

The following diagram illustrates this process. Here, data is transferred on the positive clock edges labeled 2, 3, 5, and 6.



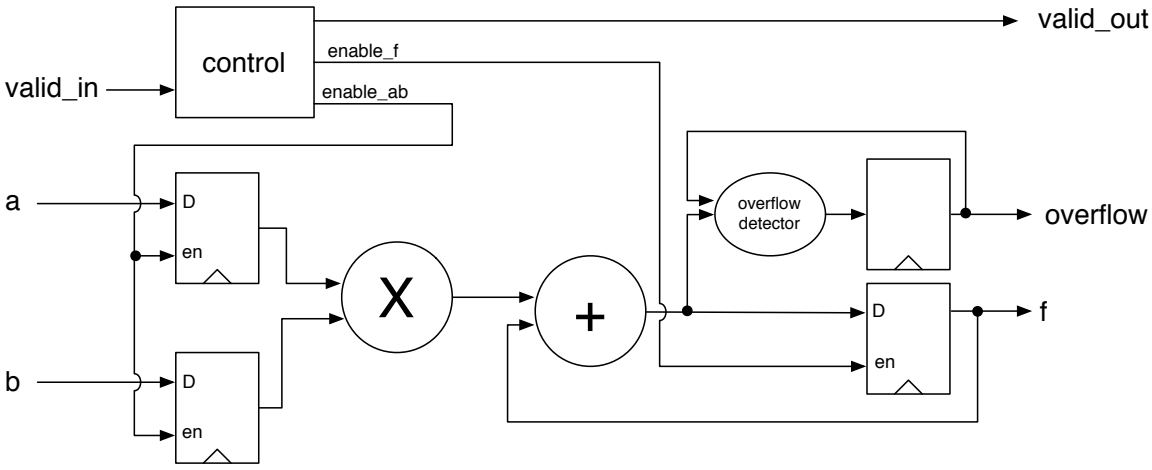
Your system will use this simple protocol for its inputs and outputs. That means your system must take as input a VALID signal and ignore the inputs anytime the VALID signal is low. Similarly, your system will produce an output VALID signal. This signal should be 1 only when a new output data word is transferred, and will be 0 when no new valid output data element is available.

Computation

Multiply and accumulate (MAC) is a basic operation used commonly 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"). Your system will look like this:



Clock and reset signals are omitted in this figure, but will be needed. The "valid_in" signal will be true when valid input data is provided on both the a and b inputs.

Note that we also have registers on the inputs a and b. Each of these three registers has an "enable" signal that will be generated by your control module. When a register's enable is 0, the register will not load any new input (simply remembering its previous value). Your control module should generate these enable signals based on the valid_in input. Similarly,

your system should produce a "valid_out" output that is 1 when your system is outputting a new data word; this signal should also be created by your control logic.

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 reset synchronously—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. Design a piece of logic that checks whether or not the system has overflowed or underflowed. Anytime either of these happens, your system must assert the "overflow" signal. Note that this signal is registered so that it will reach the system's output at the *same time* as the output value that was the result of the overflowed operation. ***Once the value has overflowed, keep the "overflow" output asserted until the system is reset.***

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

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

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. You will be evaluated on the correctness and efficiency of this system.

   I am providing you with an extremely simple testbench to make sure your system's basic timing is correct. ***This testbench is not in any way a sufficient way to test the overall correctness of your design.*** It is simply to help you get started and so to help you make sure that you are understanding the basic input/output requirement of the design.

   You can find this testbench at:
   `/home/home4/pmilder/ese507/part2_tb_simple.sv`

2. Write your own testbench that will test your module well. Make sure it is clear how to tell from your testbench's output whether or not your system worked correctly. You will be evaluated based on the correctness ***and thoroughness*** of your

testbench. Think carefully about how you are testing your system, and justify this in the report.

   a. Don't forget about the effect of the "valid_in" input when you create your testbenches. The valid_in signal could be de-asserted at any time — make sure you test that your system works correctly regardless of its behavior. Can you think of a good way to test your system with different valid_in timings?

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. Do you have any ideas of how you could have designed a more robust testbench?

   b) Explain how you detected when the system overflows, and how you tested that the overflow detection worked correctly.

   c) Report 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. When you report the critical path location, make sure you explain where in the logic the location is. (Don't just copy/paste the location given in the report—explain it in a few words.)

   d) Make graphs that show the relationships you found between clock frequency and both area and power. Explain the trends that you observed and explain why they occur. (Make graphs with frequency on the x-axis and area or power on the y-axis.)

   e) For the design you found with the maximum clock frequency, how much energy would your system consume if your system were to process a sequence of 50 cycles of input values? Assume you have to wait until the final output comes out

of the system. Additionally, assume that valid_in is asserted true for the entire time.

f) Write an expression for the best-case *energy per operation* of your system (assuming that valid_in is always true). In other words, how much energy does the system require for every multiply-and-accumulate operation, if the valid_in signal is asserted all the time?

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

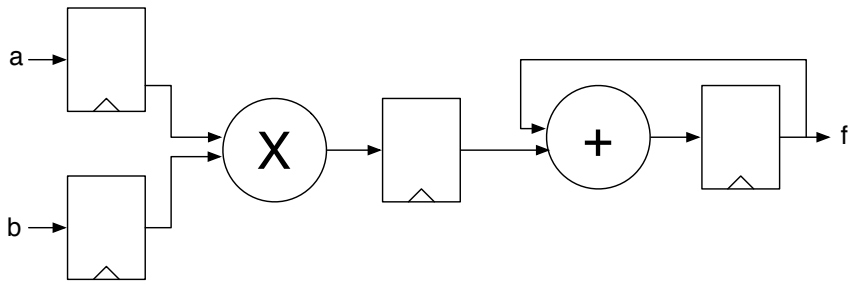
h) The directions above told you to include reset signals on the registers. Is it necessary for you to do so for the system to work correctly? For all registers? Explain.

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? How do you need to change the control logic? 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 one of the DW02 instantiations for your multiplier, when you simulate your design in ModelSim, you will need to include the *simulation module* for the pipelined multipliers. To do, simply type this command before running vsim:

```
vlog /usr/local/synopsys/syn/dw/sim_ver/DW02_mult*.v
```

Use this step to make sure your pipelined system works correctly in simulation.

(You will not need to do anything special when trying to synthesize these modules.)

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, the corresponding area and power, and the critical path location. Compare the energy-per-operation of these designs to what you calculated in Part 2.

4. In your report, answer: 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? Answer and explain in your report.

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. Include your report in the electronic hand-in with your code **(as a PDF file only)**.

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.