Stony Brook University

Project 1: MAC Unit

Issued: 9/15/21, Due: 10/4/21 4:25PM

Introduction

The objectives of this assignment are (1) to gain experience creating 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 your designs simulate correctly using ModelSim ***on the lab computers or the CAD servers***.
2. Use three different subdirectories for the code for part 1, part 2, and part 3 of the project.
3. Make sure the names and behavior of all signals match the specification in this handout ***exactly.***
4. Carefully label and document your code.

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.

Partner

If you are choosing to work with a partner (for all projects this semester), by Monday 9/20 at 11:59pm you must:

- Send an email to peter.milder@stonybrook.edu with the subject "ESE 507 Project 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)

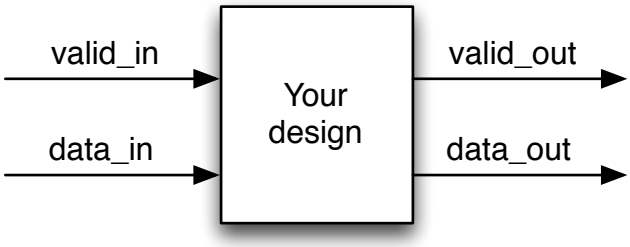
After this, you are committed to work with this partner for all three projects in the course for the entire semester.

Part 1: Multiply and Accumulate [53 points]

The goal of Part 1 is to construct a multiply-and-accumulate unit (abbreviated MAC for "**M**ultiply and **AC**umulate"). Later projects will build on this, using your MAC as a basis for larger systems.

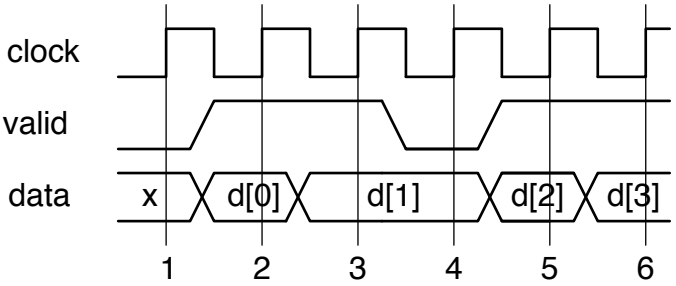
Input/Output Signaling Protocol

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 stall 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.



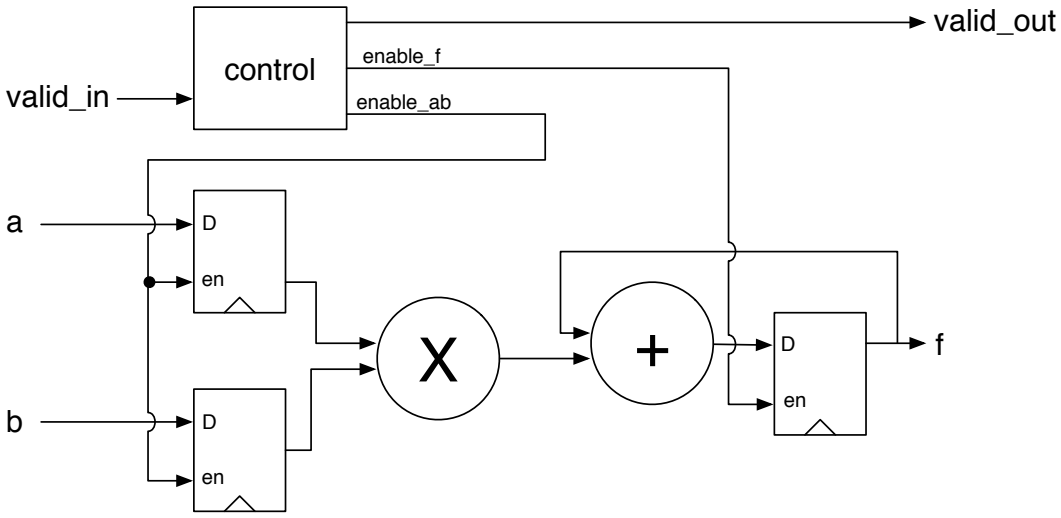
Your system will use this simple protocol for its inputs and outputs. That means your system must take as input a `valid_in` signal and ignore the inputs anytime the `valid_in` signal is 0. Similarly, your system will produce an output `valid_out` signal. This signal should be 1 only when a valid output data word is transferred and will be 0 otherwise. The `valid_out` output signal should not depend on the specific value being computed; it depends only on when a new computation is completed. (In other words, even if the new output value is the same as the previous output value, the `valid_out` signal should be asserted to indicate a new value has been produced.) Each computed output should be valid for exactly one cycle.

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 value of f. (This is called "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. You will create control logic that produces the `enable_ab`, `enable_f`, and `valid_out` signals. ***Hint: This control logic should be quite simple. It does not even require a finite state machine. Think carefully about how you can determine when to set these signals based on*** `valid_in` ***(and the clock).***

Assume that a and b are each 10-bit signed values. The output of the multiplier is a 20-bit signed value, and f is a 20-bit signed value. Assume the reset is positive-asserted (that is, when `reset==1`, the registers reset to 0). Assume all registers reset synchronously—they only reset when the "reset" signal is equal to 1 on a positive clock edge.

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. Here in Part 1, you will simply ignore over/under-flows. In the event that this happens, your system should simply continue to run and accumulate new data. (We will improve on this behavior in Part 2, below.)

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

```
module part1_mac(clk, reset, a, b, valid_in, f, valid_out);
    input clk, reset, valid_in;
    input signed [9:0] a, b;
    output logic signed [19:0] f;
    output logic valid_out;
```

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.

Use `part1_mac.sv` as the name of the file that holds your top-level module. (You may choose to include your entire design in this file or use multiple files.)

Your tasks for Part 1 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 method to test the overall correctness of your design.*** It is simply to help you get started and to help you make sure that you are understanding the basic input/output and timing requirements of the design.

   You can find this testbench at:
   `/home/home4/pmilder/ese507/proj1/part1_tb_simple.sv`

2. Write your own testbench that will test your module ***well***. Think carefully about what kind of verification strategy makes sense and will test your design thoroughly. Make sure it is clear how the user can 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 (see question 4a below).

   - 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. Adapt the scripts from HW2. Don't forget to configure the script at the top with your top module name,

clock frequency, and so on. Submit your clearly labeled synthesis report (as a plaintext file) with your project.

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)
```

If you have questions about other errors or warnings, first follow the instructions from the end of Topic 4 to use the "man" command in DesignCompiler to read the corresponding manual entries.

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) Create a test that causes the accumulator to overflow. Explain how you did it and what you observed. If you wanted the system that could detect when overflow happened, how would you do so?

   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 or a picture.)

   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 two graphs. On both, show clock frequency on the x-axis; then show area as the y-axis on one graph and power as the y-axis on the other.)

   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.

   f) Would the energy you computed in question e. change 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 2: Saturating Arithmetic [21 points]

in Part 1, you observed that your system's accumulator can ***overflow*** (or underflow). This occurs when the value accumulated no longer fits within the accumulator's register. Therefore, when the accumulated value's magnitude grows too large, the system will no longer be accurate.

In practice, there are several reasonable ways to deal with this situation. One simple approach is to size the accumulator to be "big enough" to be accommodate the largest value that you expect to be accumulated. For example, imagine that you know that your system will be used in an application where it will only be used to accumulate 128 different times before it is reset back to 0. Then, you could size your accumulator (and adder) to be $20+\log_2(128) = 27$ bits, and you can guarantee the system will not overflow unless it is used to accumulate more than 128 times.

Another approach is to detect overflow automatically. This system could have a special "overflow" output that indicates when the accumulated sum has overflowed. In this case, it would be up to the system designer to determine what to do. Overflow detection could also be combined with the previous example (using more bits in the accumulator).

A third approach that is used in some situations is called ***saturation***. Saturation means that rather than overflowing, your adder will simply limit its output to the maximum magnitude value that can fit within the given number of bits.

To illustrate the idea with an example, let's consider a slightly simpler problem: one where we have a signed 8-bit adder (used with an 8-bit register in an accumulator). The 8-bit register can hold values in the range of 8'b10000000 to 8'b01111111, or –128 to +127.

Now imagine that you are adding two numbers: 106 + 102. Obviously, the sum (208) is too large to fit within the 8-bit output. A "normal" adder would simply overflow, producing 8'b11010000 or –48. On the other hand, a saturating adder would notice that the sum would overflow, and instead output the maximum value it can represent: 127. Both sums have error, but the saturating version is closer to the ideal value than the overflowed version. The following table shows a few examples, again for our 8-bit example:

a	b	ideal a+b	a+b with overflow	a+b with saturation
106	102	208	–48	127
64	64	128	–128	127
–80	–70	–150	106	–128

In Part 2, you will replace your accumulator with a 20-bit saturating accumulator. That is, you will add logic that can detect when a sum would overflow (or underflow) and replace the value with the appropriate maximum or minimum value. Your system should behave

like the 8-bit example above, except with 20 bits. This means that the maximum value you can represent it

```
20'h7ffff = 524,287
```

and the minimum value is

```
20'h80000 = –524,288
```

Complete Part 2 by doing the following steps:

1. Make a new copy of your design from Part 1. Use a different subdirectory (I suggest `part1/` for part 1 and `part2/` for this.) Change the module name to `part2_mac` (but otherwise use the same module specification). Modify your design to perform saturation. Use `part2_mac.sv` as your top-level filename.

2. Make a copy of your Part 1 testbench and update it to test your new saturating multiply-accumulate unit. Make sure the testbench verifies the correctness of your saturation logic.

3. Use Synopsys DesignCompiler to synthesize your design, like in Part 2. Submit your clearly labeled synthesis report with your project.

   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.

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

   a. What changes did you make to the accumulator design to add saturation?

   b. Explain what changes you made to your testbench to verify that your changes were correct.

   c. Report the area, power, and critical path location for the highest clock frequency you were able to meet. Compare this result to your Part 1 design. How does your added saturation logic affect the maximum reachable clock frequency, as well as the area, power, and critical path location? For Part 2, you only need to evaluate the maximum clock frequency; you do ***not*** need to make graphs that show how area/power depend on frequency like in Part 1.

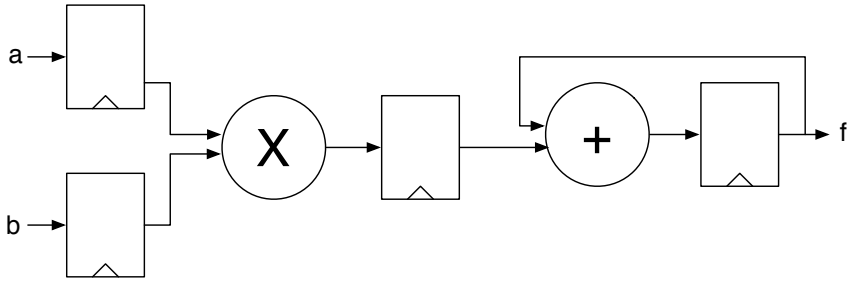
   d. How much energy does your system consume if it were to process a sequence of 50 cycles of input values? Compare this energy number to the one computed in Part 1.

Part 3: Pipelining [21 points]

For Part 3, make a new copy of your design from Part 2 in a new directory. 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 saturating 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. For this module, use filename `part3_mac.sv` and name the top-level module `part3_mac`.

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. 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. For this step, make a copy of your `part3_mac` module (from the previous step) and call it `part3b_mac` with filename `part3b_mac.sv`

   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. Determine the energy consumed by your system when processing 50 cycles of input values and compare the result to your result from Part 2.

4. In your report, answer: Which is the *best design*? Justify quantitatively how you chose the "best."

5. Would it be feasible to also add pipelining to the *adder*? If not, explain why. If so, would this create any other problems or difficulties? Answer and explain in your report.

Code and Report Submission

10 points will be awarded based on the quality of your code, comments, and report.

1. **Code**
   You will turn in a single `.zip, .tar,` or `.tgz` file to Blackboard. ***Do not use a different archive format (e.g., .rar). Seriously, please do not use any archive format except .zip, .tar, or .tgz or you will be deducted points.***

   This compressed file should hold all of the files from your project. I will be testing your designs using my testbenches, so it is very important that you stick to the specification closely. I will test your designs using the ECE grad lab computers so make sure everything runs correctly ***there***.

   Do not turn in things like ModelSim "work" directories or gate-level Verilog produced by synthesis. Please only submit your actual code.

2. **Synthesis Reports**

   Include the DesignCompiler synthesis report (in plaintext format) for each design you synthesized. Make sure these reports are clearly labeled to indicate which part of the project they are for.

3. **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)**. If you worked with a partner, at the end of your report, explain each partner's contribution to the project. (If you worked alone obviously you can skip this.)

4. **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.

   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.

Please, only use .zip, .tar, or .tgz files for your archive, and use PDF for your report. If you use other formats I will be unable to open your work on the lab computers, and you will lose points. Do not turn in things like ModelSim "work" directories or gate-level Verilog produced by synthesis.