EECS 470 Project #2

- This is an individual assignment. You may discuss the specification and help one another with the (System)Verilog language. Your solution, particularly the designs you submit, must be your own.
- Due at 11:59pm ET on 10th February, 2021. *Late submissions are not accepted.*

---

1 Introduction

We've mentioned synthesis several times already in the lab and lecture. Synthesis is an exponentially complex process, and so it can take a very long time on larger projects, like the one you'll be doing at the end of the semester. To mitigate this, we can synthesize submodules and then include them as black boxes in the final project, which simplifies the synthesis problem. This is particularly useful for synthesizing the caches in your final project, which can reduce the synthesis time by half or more.

We will use a pipelined multiplier as an example of this process. This multiplier is actually the one you'll be using in your final project, so you should use this as a chance to get very familiar with it.

Finally, we will be using this multiplier as part of a finite state machine design problem, which is intended to give you additional practice writing Verilog in reasonable style. The style we recommend for finite state machine design will be presented in the lab.

2 Hierarchical Synthesis

2.1 Concept

Synthesis takes a behavioral level design and turns it into a structural level design. Specifically, the synthesis tool replaces larger operations, i.e. multiplication or addition with standard designs for modules, and attempts to build larger logic components out of a set of standard cells (our standard cells are in the `lec25dscc25.v` file included by your `Makefile`). Doing this optimally is an NP-hard problem, so it is often impossible in practice. In this class, it is generally merely very time consuming, meaning that your final project will take anywhere between six hours and one day to synthesize.

To mitigate this problem, we can synthesize large submodules in a design individually and then include these syntheses as black boxes in the final design synthesis. Given that the original problem was NP-hard, we know that it would have been exponentially complex, where the quantity of interest is the size of a design, measured in something like the number of logic elements (standard cells, LUTs, transistors, etc.) That means that simplifying to only somewhat fewer elements still shortens the time to find a solution significantly.

2.2 Pipelined Multiplier

We have provided you with a pipelined multiplier, found in `pipe_mult.v`, to which we will apply this concept. The multiplier does multiplication in stages, somewhat like you would have learned to carry out multiplication in elementary school. It multiplies the first 8 bits of the multiplier with the whole multiplicand in one clock cycle, then the next 8 bits of the multiplier against a shifted multiplicand, and so on to get 8 partial products. Summing those gives us the desired multiplication. This means that each multiplication will take 8 clock cycles. Each partial product is created by a separate multiplier stage, which can be found in the `mult_stage.v` file.

Your first assignment will be to find a reasonable clock period (within 2ns of the lowest possible) at which the 8 stage multiplier we've provided you can be synthesized with slack met. For example, if you find that the design does not synthesize at a clock period of 4 ns, but it does synthesize with a clock period of 5.5 ns, then 5.5 ns must be within 1.5 ns of the shortest possible clock period. We've already setup the `.tcl` scripts to synthesize the two modules in this design for you. Note the `set_dont_touch` command in the `mult.tcl`

script. This command tells the synthesis tool to treat the module as a black box and to not optimize it further.

To find the clock period, you will need to change the clock period in the `mult_stage.tcl` file until you reach the best clock period that meets all slack. Once you've found that, the total clock period of the multiplier will be that stage clock period plus the additional combinational delay from the interconnects in the higher level `mult` module. This means that the clock period in the `mult_stage.tcl` file will be different from the one found in the `mult.tcl` file.

Your second assignment is to modify the pipelined multiplier we've provided to work as both a 4 stage multiplier and as a 2 stage multiplier. You can either do this by copying the files and having separate 2, 4 and 8 stage multipliers or by figuring out a combination of preprocessor macros or parameters that set pipeline depth. Whichever method you choose is fine for this project, but having the parameterized pipeline depth will be extremely helpful for the final project. Once you have the other two multipliers, you will need to find the best clock periods that they can each achieve. How do all three clock periods compare? Does this conform to your expectations?

3 Integer Square Root

Now, you will need to create a module that uses the multiplier that we supplied, the 8 stage multiplier. You will be writing a module to compute the integer square root of a 64-bit number. It will generate a 32-bit number that is the largest integer that is not larger than the square root of the number provided. For example, the integer square root of 24 is 4.

The module declaration is as follows:

```
module ISR(
    input                   reset,
    input           [63:0]  value,
    input                   clock,
    output logic    [31:0]  result,
    output logic            done
);
```

It should operate as follows:

- If `reset` is asserted during a rising clock edge (synchronous reset), the `value` signal is to be stored.
- If `reset` is asserted part way through a computation, the result of that computation is discarded and a new `value` is latched into the module.
- When the module has finished computing the answer, the output is placed on the `result` line and `done` line is raised on the same cycle.
- It must not take more than 600 clock cycles to compute a result (from the last clock that `reset` is asserted to the first clock that `done` is asserted.)

We do not suggest that you pipeline this module. You will likely need to perform something like a binary search to find the result a simple algorithm is as follows:

Algorithm 1 Integer Square Root

```
1: procedure ISR(value)
2:     for i ← 31 to 0 do
3:         proposed_solution[i]←1
4:         if proposed_solution^2 > value then
5:             proposed_solution[i]←0
6:         end if
7:     end for
8: end procedure
```

Note that loops do not have a direct hardware equivalent. What hardware design technique lets us implement a procedure like this?

In addition to writing this module, you will need to write a testbench for it. This testbench should probably test specific corner cases, random testing and the short loops. Your testbench should print either `@@@Passed` or `@@@Failed`.

Once you have the module written and tested, synthesize it. This will probably take several minutes at least.

4 Comprehension Questions

Answer the following questions, and submit the answers in a plaintext file called `answers.txt` along with the rest of your project.

1. Consider the multiplier supplied with this project.
   (a) What is the cycle time achieved when you synthesized our multiplier?
   (b) What does this mean the total latency is for a multiplication?
2. Answer question 1 (parts a and b) for the two multipliers you created.
3. Consider the relative values of the answers you found to questions 1 and 2. Do these seem reasonable? Why or why not?
4. What is the clock period achievable with the module you wrote in section 3?
5. How long would it take for your module to compute the square root of 1001 given the cycle time of question 4? Would you expect a performance gain or penalty if you used your 2 stage multiplier?

5 Submission

After you are confident in your solution, make sure you have the following files in your directory

1. `ISR.v`
2. `test_ISR.v`
3. `answers.txt`

Now, go up one directory (`cd ..`) and run:

```
/afs/umich.edu/user/j/b/jbbeau/Public/470submit -p2 <directory>
```

Note that the script takes a directory name as an argument, not an absolute path, so you must run it from one level above your project 1 directory.

At this point, you should see something similar to:

```
Submitting files in project2/
--- submitting project2/
--- submitting project2/Makefile
--- submitting project2/ISR.v
--- submitting project2/test_ISR.v
--- submitting project2/answers.txt
Submitted.
```

If there is a problem, it will be printed to the screen. Shortly after submission you should receive an email telling you if your submission passed the basic tests. This is not an autograder, so it's simply telling you that your design built successfully on the grading machine and that the module declarations looked approximately correct. If your submission didn't pass, copy the error message into an email to the course staff, and we will attempt to help.