EECS 470 Project #2

Note:

- This is an individual assignment. While you may discuss the specification and help one another with the SystemVerilog language, your solution – particularly the designs you submit – must be your own.
- This project is due by **Monday, September 15th**
- This project will have you submit answers to multiple comprehension questions. Place these in the provided `answers.txt` file.

1 Introduction

In this project you will be evaluating a pipelined multiplier with parent and child modules (`mult` and `mult_stage` respectively) to examine how changing the complexity of a child module impacts the clock period in hierarchical synthesis.

You will then use the multiplier as part of a finite state machine design problem for an integer square root algorithm and testbench. This 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

We've mentioned synthesis several times already in the lab and lecture. Synthesis is the process of converting a behavioral verilog design into a structural design which could be taped-out on real silicon.

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 the Makefile).

Doing synthesis optimally is an NP-hard problem, which makes optimal synthesis impossible in practice. In this class, we merely aim for good-enough synthesis, which is merely very time consuming. While projects 1 and 2 will often synthesize in under 10 minutes, your final project may take multiple hours 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 and 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.), simplifying to fewer elements should shorten the time to find a solution!

In synthesized modules, we generally focus on timing rather than area in this class, and use the **slack** metric: the clock period minus setup time minus the longest signal path from one clocked register to another. If slack is negative, a signal may still be changing when the register starts reading it at the next clock edge, leaving incorrect values floating and causing potential undefined behavior (scary!).

If slack is positive or zero, we consider the slack *Met*, but if it's negative, we consider the slack *Violated*. You will see this slack status in the output of synthesis in the `synth/mult.rep` file. The image below also illustrates a timing diagram with positive slack (waveform a) and negative slack (waveform b).

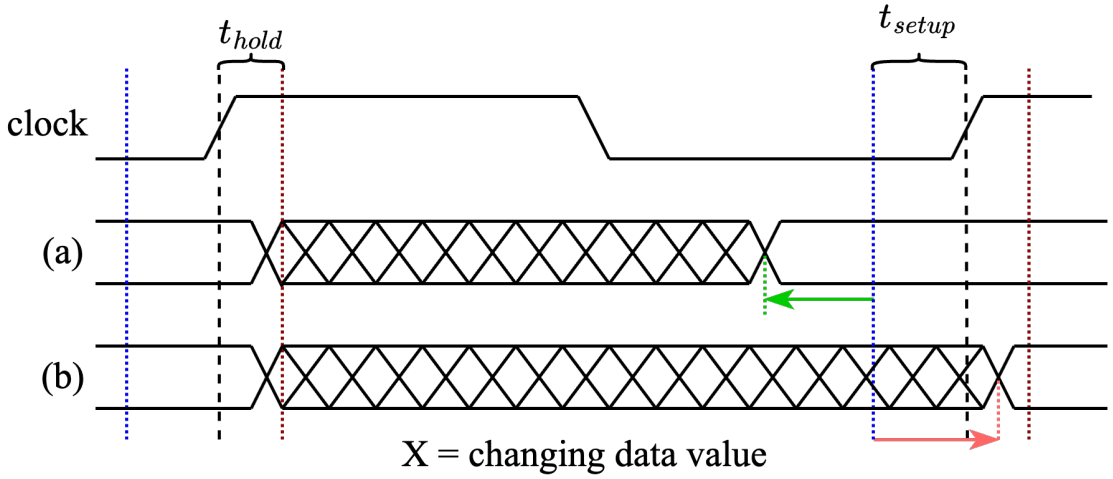

Figure 1: Timing diagram where (a) has positive slack and (b) has negative slack

3 Pipelined Multiplier

We have provided an arbitrary-stage multiplier as two modules and a header: the parent module `mult`, in `mult.sv`, the child module `mult_stage`, in `mult_stage.sv`, and the header defining the number of stages in `mult_defs.svh`.

We have also provided a testbench (`mult_test.sv`) and a Makefile set up for hierarchical synthesis. The Makefile has new variables `CHILD_MODULES` and `CHILD_SOURCES` for the names and source files of the child module and it has the variable `DDC_FILES` for the design compiler data files that will store the synthesized child module for re-use.

The Makefile starts ready to test the multiplier, feel free to run `make sim` now to compile and run the testbench for simulation.

You can synthesize the `mult` module with `make synth/mult.vg`, and can view the slack with `make slack` (which just greps for "slack" in the generated `synth/mult.rep` file).

Hold off on running any synthesis commands for now.

3.1 Multiplier Interface

The pipelined multiplier and its stages have the following interfaces:

```
module mult (
    input clock, reset,
    input [63:0] mcand, mplier,
    input start,

    output [63:0] product,
    output done
);
```

Figure 2: The parent module: `mult`

```
module mult_stage (
    input clock, reset, start,
    input [63:0] prev_sum, mplier, mcand,

    output logic [63:0] product_sum, next_mplier, next_mcand,
    output logic done
);
```

Figure 3: The child module: `mult_stage`

`mult`, multiplies its two inputs – the "*multiplier*" and "*multiplicand*" – in stages similar to how you might have learned to multiply in elementary school. For the 8-stage module, it starts by multiplying the first 8 bits of multiplier with the entire multiplicand in one clock to create one partial product, then shifts the multiplicand and multiplies the next 8 bits of the multiplier to make a new partial product. These partial products are summed with the previous sum in each stage, and will output our final product after 8 stages.

`mult` will read the values of the multiplier and multiplicand on the cycle where `start` is true, and will set `done` true when it's finished. While `done` is true, the value of `product` is the final product of the multiplication. The internal `mult_stage` modules used by `mult` also use start and done signals, and `mult` forwards the done signal of a previous stage to the start of the next stage.

3.2 Fully Pipelined?

Any pipelined module may or may not be *fully-pipelined*. A fully-pipelined module can accept one input on each cycle, and will calculate each of the outputs in separate pipeline stages concurrently, returning the results in-order.

Based on the description above and the provided implementation, is our multiplier fully-pipelined?

You should open and read the `mult` and `mult_stage` modules and answer the following question in one to two sentences in the `answers.txt` file.

> **Question 1.** Is our multiplier fully-pipelined? Can it accept one input on each cycle and compute the outputs concurrently in separate cycles? Why or why not?

3.3 Finding the Minimum Clock Period

Your first task is to find the minimum clock period where the 8-stage multiplier can be synthesized with slack met.

The clock period is defined in the Makefile and starts at 10.0ns. Decrease it and re-synthesize the module (`make -B synth/mult.vg CLOCK_PERIOD=9.0`) until slack is violated, then increase it until you've found the lowest period for which slack is met.

The `-B` in the make command tells make to unconditionally *Build* the testbench, since we don't alter any files to change the clock period.

You should find a value within 1ns of the lowest possible time (if the exact value is 10.5ns, 9.5ns through 11.5ns will be accepted). Put values you find in the `answers.txt` file.

Your second task is to repeat this with both a 4-stage and 2-stage version of the multiplier. Edit the `STAGES` macro in `mult_defs.svh` and re-synthesize the modules to get the minimum clock periods.

Answer these four questions in the `answers.txt` file:

> **Question 2.** For the 8-stage multiplier, what are the minimum clock period and overall time for a multiplication (in ns)?

> **Question 3.** For 4-stage?

Question 4. For 2-stage?

Question 5. What patterns in the clock period and overall time do you see as we decrease the number of stages? Why would you expect this pattern?

Reset the multiplier to 8 stages (and run `make nuke`) for use in the ISR module.

4 Wonka's Square Chocolate Bars

After elegantly solving his parking troubles, Willy Wonka has tasked you with solving a problem on his chocolate assembly line. His newest product, Wonka's RISC-Y Chips™, must be a perfect square of chocolate. Although quick at setting and wrapping chocolate, the Oompa Loompas on the RISC-Y Chips assembly line are slow at calculating the correct dimensions of the chocolate square given a mass of chocolate. You are to design a system to help the Oompa Loompa's calculate the dimensions as fast as possible.

4.1 Integer Square Root

You will now both design and test 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 not larger than the square root of the number provided. For example, the integer square root of 24 is 4.

You should implement it using the following binary search algorithm:

Algorithm 1 Integer Square Root

```
procedure ISR(value)
    result[31:0] ← 0
    for i ← 31, 30, ..., 0 do
        result[i] ← 1
        square ← result * result
        if square > value then
            result[i] ← 0
        end if
    end for
end procedure
```

Your module should have the following declaration:

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

Figure 4: The ISR Module

And should operate as follows:

- If `reset` is asserted during a rising clock edge, the `value` signal is to be stored and the module should begin computing the ISR of `value`.

- If `reset` is asserted part-way through a computation, the `value` signal should be overwritten and the module should restart computation, discarding any previous results.
- When the module has finished computing the result, the `done` signal should be set high and the `result` output should contain the integer square root of the stored value. `done` should be high for one full 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). (It can be done in many fewer cycle)

Note that for-loops like the one in the algorithm do not have a direct hardware equivalent. You will need to use a finite state machine to replicate this procedure.

Multiplication within the `ISR` module should be accomplished using the `mult` module, not by using the * operator.

4.2 Write a testbench

In addition to writing this module, you will need to write a testbench for it. This testbench should test specific edge cases and random values (similar to lab 2).

Your testbench must `$display` "`@@@ Incorrect`" for an incorrect ISR implementation and must `$display` "`@@@ Passed`" for a correct implementation.

You should use the provided multiplier testbench as a starting point for yours. It already contains a Verilog "task" for waiting on a module's `done` output to be true, which will be very useful for testing the ISR module.

We have provided 3 obfuscated, incorrecct ISR modules as `ISR_buggy1.vg`, `ISR_buggy2.vg`, and `ISR_buggy3.vg`. Your testbench should be able to catch the bugs in each of them. The Makefile has been extended with new targets to compile and run these, and the `make nuke` target will not delete them. The new targets are:

```
make build/buggy<1-3>.simv       <- compiles your ISR_test.sv with a buggy ISR module
make buggy<1-3>.sim  <- runs a buggy ISR module on your ISR_test.sv
make buggy<1-3>.verdi      <- run the buggy executable in verdi
```

Figure 5: New make targets for testing the buggy ISR modules

4.3 Find the minimum clock period

Once you have the module written and tested, synthesize it and find the minimum clock period of your module. Answer the final two questions in the `answers.txt` file:

Question 6. What is the minimum clock period for your ISR module (with the 8-stage multiplier)?

Question 7. How long (in ns) would it take your module to compute the square root of 1001 given the period from question 6? Would you expect a performance gain or penalty if you used the 2-stage multiplier?

Complete the table below in the template in `answers.txt`. In the cycles column, "Setup" refers to the time between reset and the first multiplication occurring, "Compare" is the number of cycles between multiplications, and "Done" is the number of cycles between the last multiplication finishing and the final "done" output being asserted.

5 Comprehension Questions

Answer the following questions and submit your answers in the provided `answers.txt` file:

Question 1. Is our multiplier fully-pipelined? Can it accept one input on each cycle and compute the outputs concurrently in separate cycles? Why or why not?

Question 2. For the 8-stage multiplier, what are the minimum clock period and overall time for a multiplication (in ns)?

Question 3. For 4-stage?

Question 4. For 2-stage?

Question 5. What patterns in the clock period and overall time do you see as we decrease the number of stages? Why would you expect this pattern?

Question 6. What is the minimum clock period for your ISR module (with the 8-stage multiplier)?

Question 7. How long (in ns) would it take your module to compute the square root of 1001 given the period from question 6? Would you expect a performance gain or penalty if you used the 2-stage multiplier?

Complete the table below in the template in `answers.txt`. In the cycles column, "Setup" refers to the time between reset and the first multiplication occurring, "Compare" is the number of cycles between multiplications, and "Done" is the number of cycles between the last multiplication finishing and the final "done" output being asserted.

	Cycles				**Clock**	**Total**
	Setup	**Multiply**	**Compare**	**Done**	**Period**	**Time**
2-cycle Multiplier						
8-cycle Multiplier						

6 Submission

To submit your project to the EECS 470 autograder, upload your solution files to the `main` branch of your GitHub repository and run the project submission script for project 2:

```
/usr/caen/misc/class/eecs470/submit 2
```

Soon after your submission you will receive an email with a summary of the public output of the autograder. This will contain only the public results of correctness tests and whether the autograder encountered any errors and is not representative of your final grade.

Email the instructors or create a private Piazza post to debug any issues with the autograder.

The following files will be graded:

- `answers.txt`
- `ISR.sv`
- `ISR_test.sv`