CS M152A - Lab 2 Report

Adam Vuilleumier
UID: 305098071
Lab Section 6 (Rajas Mhaskar)

1. Introduction

In this laboratory, my goal was to utilize my knowledge of clocks, counters, and if statements to create clock outputs of different frequencies, duty cycles, and count values. To do this, I created the Verilog file clock_gen.v, which consisted of four different submodules that handled the following: division by powers of 2, even division, odd division, and creating a repeating series using a strobe.

In addition to my clock_gen program, I also completed the other verification tasks listed in the project specification. These verification tasks often gave me great insight with regards to which methods I would have to use in the design tasks.

Before I began work on this project, I needed to familiarize myself with the terminology and concepts of clocks. Often, the spec asked us to manipulate the duty cycle and frequency of a clock. A clock's duty cycle is the percentage of time that it spends at a high voltage on each cycle. For example, if a clock spends 10 ns at high voltage before going to a low voltage for 90 ns, it is said to have a 10% duty cycle. The frequency of a clock, on the other hand, is the reciprocal of the clock's period (the amount of time it takes for a full cycle from 1 to 0 and back to 1 again). When we create a "divide by X" clock, we are dividing the frequency by X, which is the same as multiplying the time period by X.

Below is a list of each design and verification task, a description of their functionalities, and (for the verification tasks) the relation to clock_gen's logic. Each of them uses the input clock clk_in and the signal rst, which, when active, resets the clock to its original state.

Design/Verification Tasks:
1 - Create a divide by 2, 4, 8, and 16 clock using the four bit values of a counter.
2 - Create a divide by 32 clock that flips its output on a counter overflow. This gave me the idea to flip the output and reset the counter back to 0 once the counter reached the desired value. I ended up using this method for task 3.
3 - Create a divide by 28 clock with a 50% duty cycle (implied).
4-6 - Create 33% duty cycle clocks that trigger on the rising and falling edges of the input clock. Also create a wire that takes the logical or of these two clocks. This task gave me the idea to use the or operator to merge the outputs of two clocks, which ended up being useful when I had to create a divide by odd clock in task 7.
7 - Create a divide by 5 clock with a 50% duty cycle.
8 - Create a 50% duty cycle divide by 200 clock that operates at 500 kHz by using a divide by 100 pulse. This task got me used to strobes/pusles and to using the output of one clock as the input of another, which were also helpful in task 9.
9 - Create an 8-bit counter that increments by 2 each cycle and subtracts 5 every fourth cycle.

2. Design Description:

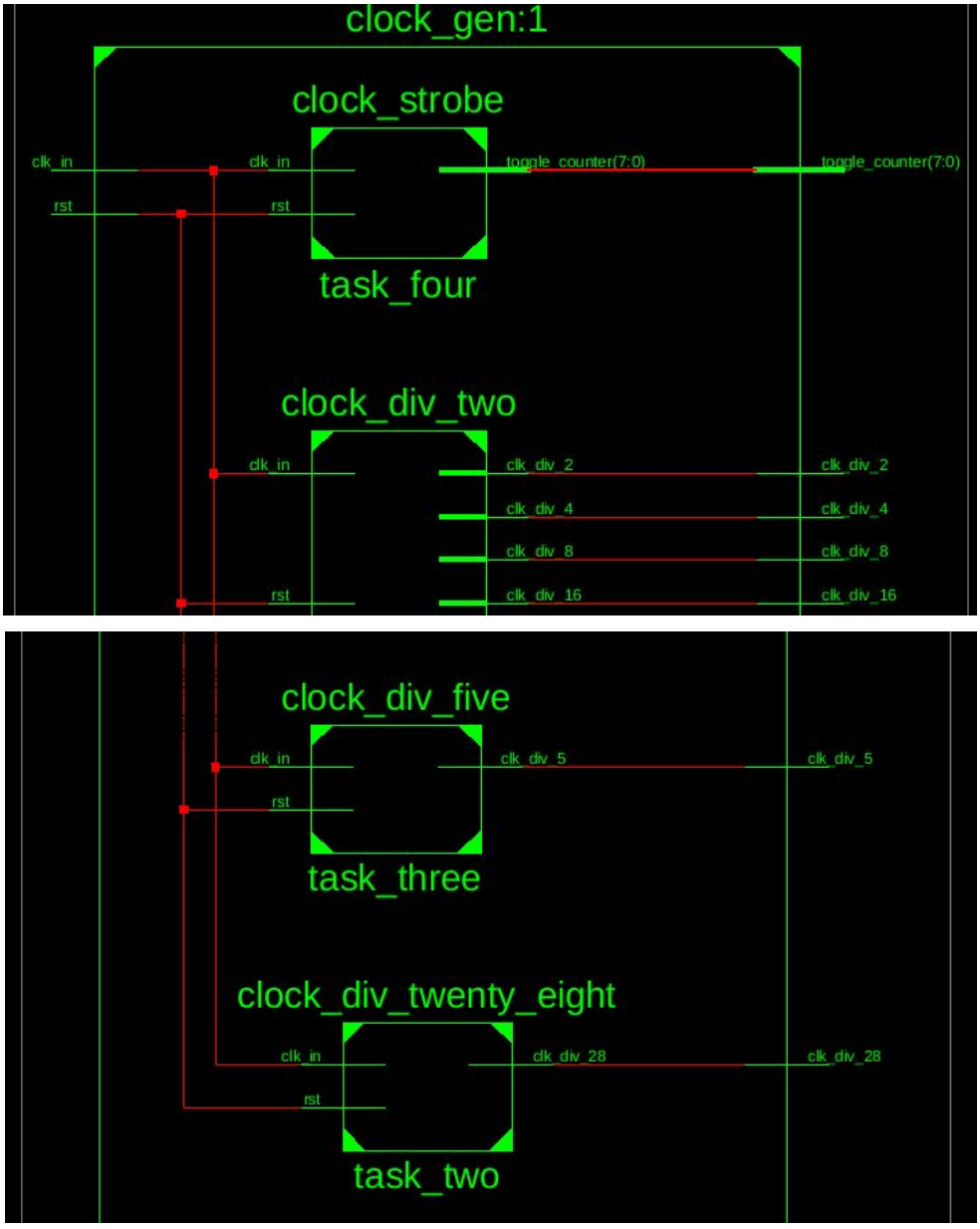

Fig 2.1: Top-Level Module Design Schematic

This diagram shows the four submodules of clock_gen. Each submodule takes in the parameters clk_in and rst and outputs some sort of clock - or counter in the case of clock_strobe. I will explain the design of each submodule more fully later in this section.

```
--------------------------------------------------------------------------------
HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                                   : 6
 3-bit adder                                           : 2
 32-bit adder                                          : 1
 4-bit adder                                           : 2
 8-bit addsub                                          : 1
# Registers                                            : 7
 1-bit register                                        : 1
 3-bit register                                        : 2
 32-bit register                                       : 1
 4-bit register                                        : 2
 8-bit register                                        : 1
# Comparators                                          : 2
 3-bit comparator greater                              : 2
# Multiplexers                                         : 1
 8-bit 2-to-1 multiplexer                              : 1

================================================================================
```

Fig 2.2: Synthesis Report

The synthesis report tells us all the specific adders, registers, muxes, and comparators that would be needed to create the physical FPGA. However, since this project is purely based on clock simulation, it would not be useful to actually build the hardware. Nonetheless, this report still helps us break down and understand all the smaller steps of the implementation.

```
Slice Logic Utilization:
  Number of Slice Registers:                 22 out of   4,800    1%
    Number used as Flip Flops:               22
    Number used as Latches:                   0
    Number used as Latch-thrus:               0
    Number used as AND/OR logics:             0
  Number of Slice LUTs:                      19 out of   2,400    1%
    Number used as logic:                    18 out of   2,400    1%
      Number using O6 output only:           10
      Number using O5 output only:            1
      Number using O5 and O6:                 7
      Number used as ROM:                     0
    Number used as Memory:                    0 out of   1,200    0%
    Number used exclusively as route-thrus:   1
      Number with same-slice register load:   0
      Number with same-slice carry load:      1
      Number with other load:                 0

Slice Logic Distribution:
  Number of occupied Slices:                  8 out of     600    1%
  Number of MUXCYs used:                      4 out of   1,200    1%
  Number of LUT Flip Flop pairs used:        20
    Number with an unused Flip Flop:          4 out of      20   20%
    Number with an unused LUT:                1 out of      20    5%
    Number of fully used LUT-FF pairs:       15 out of      20   75%
```

Fig 2.3: Map Report

According to the map report, there are 22 slice registers (all of which are used as flip flops), and 19 slice LUTs (18 of which are used for logic and one used as a route-thru) in the final FPGA.

My submodule **clock_div_two** implements a simple 4-bit counter, a, which increments by 1 every positive edge of the input clock. To create a divide by 2^n clock from this counter, all I needed to do was assign the clock wire to a[n-1]. This method works because the second bit, for example, only changes once every two clock cycles, making it a perfect divide by four clock output.

My submodule **clock_div_twenty_eight** uses a counter which is reset to zero every time it reaches 13. When this counter is reset, the output clock also flips its value, making it so that this value changes every 14 input clock ticks. This generates a divide by 28 clock.

My submodule **clock_div_five** uses two always blocks to create two counters that each reset at 4. One of these counters counts at the positive edge of each input clock cycle, and the other counts on the negative edge. If either of these counters is greater than 2, the output is 1. This simulates a value change at 2.5 (which is half the value of 5, our desired dividing factor), and thus is the same as a divide by 5 clock.

My submodule **clock_strobe** uses a counter, c, which starts at the value 1. If the value of c is divisible by 3, it decrements toggle_counter by 5, or else it increments toggle_counter by 2. I originally implemented this module with two always blocks: the first created a strobe pulse every four input clock cycles, and the second used this strobe to decide when to increment and when to decrement the toggle_counter. However, I realized that the same effect could be reproduced more efficiently by just checking the counter for divisibility by 3.

3. Simulation Documentation

Fig 3.1: Verification Task #2

My divide by 32 clock has a frequency that is 32 times smaller than the input clock.

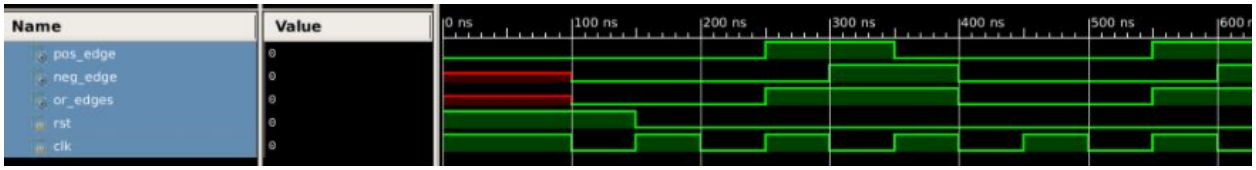

Fig 3.2: Verification Tasks #4-6

The waveform or_edges combines the above two waveforms, which are 33% duty cycle clocks that execute on the positive and negative edges, respectively, of the input clock.

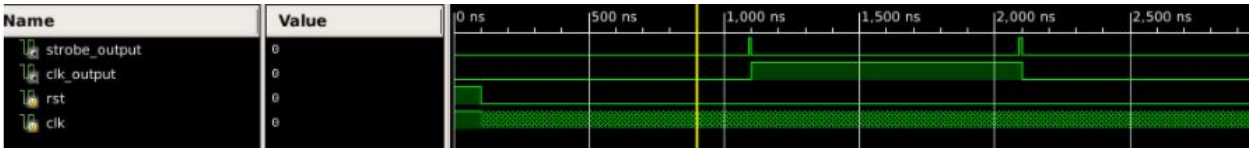

Fig 3.3: Verification Task #8

The above waveforms are the divide by 100 pulse (which uses a 1% duty cycle), and the divide by 200 50% duty cycle clock that relies on it.

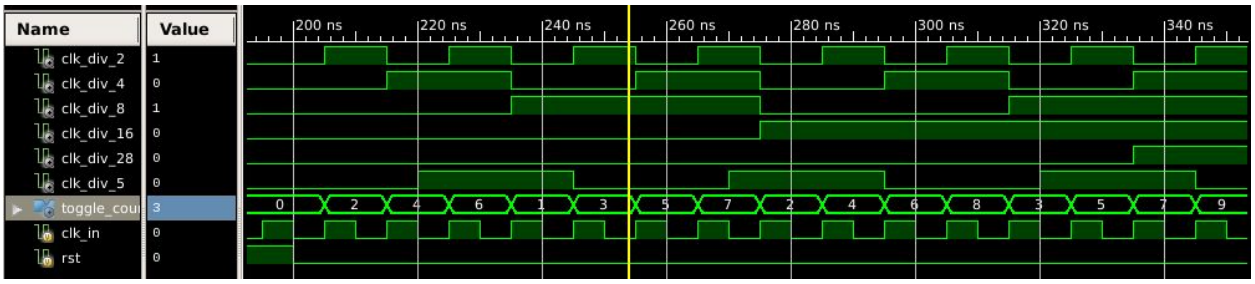

Fig 3.4: Clk_Gen Waveforms (Design Tasks)

These waveforms are for the divide by X clocks that I generated from the design task descriptions. Each of them has a frequency that is X times smaller than the input clock. Toggle_counter is an exception, as it is a counter that holds different integer values at different times as specified by the heuristic discussed in the lab spec.

4. Conclusion

By the end of the lab, I had successfully created clocks of a wide range of frequencies and duty cycles. To do this, the use of counters was integral, and I learned much about how to use these counters to get the desired output for my clocks. Basically, if you design a system where the module can recognize when a counter reaches a desired value and you flip the output at that value, you can create clocks of almost any frequency.

During the course of this lab, I struggled again with the differences between the types wire and reg, and it led to many errors in my code. A work around that I discovered is that if you want an output to be a wire but you want to change it within an always block, you can create a temporary reg variable, do all your always block assignments on it, and then assign this temp variable to your output wire. Once I learned this trick, I had more freedom with what kinds of operations I could do on all my variables.

I would like to thank my TA, Rajas Mhaskar for all his help on this lab, and his diligent instruction each week in lecture.