**Lab 2 Project Report**

Ethan Keshishian

CS M152A Lab1

Introduction:

The goal of Lab 2 is to implement 9 tasks that generate various clock waveforms. This is done with Verilog and the Xilinx software, which helps write the code as well as generate the waveforms. The design has a module called clock\_gen at the top level, and multiple submodules that help implement each task. This keeps the code for each task separate. There is a submodule for each task except tasks 4, 5, and 6, which are grouped together under one submodule. This is in line with the example design given in the lab document. The names and designs of both module and submodule match the provided design for simplicity. My final implementation handles each task and has waveforms that match the solution waveforms. Additionally, the waveforms start at the same time as the example waveforms.

Module design:

The top module is a module named clock\_gen that handles all nine tasks be delegating them to submodules. The top module takes in the clk\_in and rst variables which handle sequential logic, and outputs 12 variables. Within clock\_gen, there are seven submodules defined to handle the tasks, and each one of these accepts the clk\_in, rst, and other output variables it will modify. These seven submodules are implemented next.

1. module clock\_div\_two(clk\_in, rst, clk\_div\_2, clk\_div\_4, clk\_div\_8, clk\_div\_16)

This is the first module that handles task 1. It uses clk\_in and rst as input, and outputs clk\_div\_2, clk\_div\_4, clk\_div\_8, and clk\_div\_16.

This module uses the clock to create waveforms that are active in frequency slower than the clock. For example, clk\_div\_2 divides the clock’s frequency in half. To implement this, there is a 4-bit counter which is implemented with an always block and an internal register (called a). If rst is enabled, the counter remains at 0. Otherwise, it will increment the register by 1 with each clock cycle. This counter has no upper bound; it will overflow and restart one cycle after reaching 4’b1111. The outputs of this module are wires. The clock output is obtained by assignment to each bit of the internal register used by the counter. clk\_div\_2 is a[0], clk\_div\_4 is a[1], clk\_div\_8 is a[2], and clk\_div\_16 is a[3].

1. module clock\_div\_thirty\_two(clk\_in, rst, clk\_div\_32)

This is the module that handles task 2. It uses clk\_in and rst as input, and outputs clk\_div\_32.

This module divides the clock by 32. It similarly uses a 4-bit counter which is implemented with an always block and an internal register again called a. However, in this case, the bits cannot be assigned as before. The output is treated as a register, and the value of the register should flip every time a overflows. In order to accomplish this, a is assigned a default value of 0, and there is if statement that flips the value of a any time a is equal to 4’b111.

1. module clock\_div\_twenty\_six(clk\_in, rst, clk\_div\_26)

This is the module that handles task 3. It uses clk\_in and rst as input, and outputs clk\_div\_26.

This module divides the clock by 26. It uses a 4-bit counter with an always block and has an internal register, a. This case is similar to task 2, but the clock has to manually be reset to 0 when it has counted 26 times. In order to accomplish this, the always block that contains the counter has been slightly modified and clk\_div\_26 is set to 0 at first. If the value of a is equal to 12, then clk\_div\_26 is inverted, and a is reset to 0. This is verifiable with the waveform.

1. module clock\_div\_three(clk\_in, rst, clk\_div\_3, clk\_pos, clk\_neg)

This is the module that handles tasks 4, 5, and 6. It uses clk\_in and rst as input, and outputs clk\_div\_3, clk\_pos, and clk\_neg.

This module handles the case where the clock is divided by 3. This is challenging, because 3 is an odd number, and this splits the clock’s positive and negative edges. To handle it, there are two 4-bit counters, a and b, and they are responsible for handling the positive and negative edges, respectively. The positive edge is implemented with the a counter, similar to before. If the value of a is 2, it resets to 0. If a is equal to 1, clk\_pos is 1, otherwise it’s 0. This creates a waveform that is active for 1/3 of the time. clk\_neg is obtained in an identical always block, except it activates on the negedge instead of posedge, and it uses the b register as its counter. This also creates a waveform that is active for 1/3 of the time. Since it’s activated on the negedge instead, it’s offset from the clk\_pos waveform. Both are active at the same time 1/6 of the time. Finally then, the clk\_div\_3 wire is obtained by calculating clk\_pos | clk\_neg. Since they are offset, they are active for 1/3 + 1/3 – 1/6 of the time, which is equal to 1/2 of the time. They create a waveform that divides the clock’s frequency by 3.

1. module clock\_div\_five(clk\_in, rst, clk\_div\_5)

This is the module that handles task 7. It uses clk\_in and rst as input, and outputs clk\_div\_5.

This module handles the case where the clock is divided by 5. This is slightly more complicated than the divide by 3 case. In this case, there is again two 4-bit counters, a and b. There are also internal variables clk\_pos and clk\_neg initialized to 0. In the a always block which runs on the posedge, the counter is reset when a is equal to 4. If a is equal to 2 or 3, clk\_pos is equal to 1. Otherwise, it’s equal to 0. This creates a variable that’s active 2/5 of the time. Similarly, the second counter is implemented with an always block that activates on the negative edge of clk\_in. It also resets when b is equal to 4. If b is equal to 2, clk\_neg is 1. Otherwise, it’s equal to 0. This creates a variable that’s active 1/5 of the time and slightly offset. This offset means both clk\_pos and clk\_neg are active 1/10 of the time. Finally, the clk\_div\_5 wire is assigned to clk\_pos | clk\_neg. Mathematically this means they are active 2/5 + 1/5 – 1/10 of the time, which is equal to 5/10 or 1/2 of the time.

1. module clock\_pulse(clk\_in, rst, clk\_div)

This is the module that handles task 8. It uses clk\_in and rst as input, and outputs clk\_div as output.

This module handles the case where the clock is divided by 200. To handle this, I use a 7-bit counter to count to 100. The implementation is similar to the others, except register a is larger. If a is equal to 99, then a is reset to 0. If a is equal to 1, an internal variable called flip\_bit is equal to 1. Otherwise, flip\_bit is equal to 0. Then, in a separate always block that also activates on the positive edge of clk\_in, clk\_div is being inverted when flip\_bit is equal to 1. This give the correct output that matches the waveform.

1. module clock\_strobe(clk\_in, rst, toggle\_counter)

This is the module that handles task 9. It uses clk\_in and rst as input, and outputs toggle\_counter as output.

This module counts up by 3 every three cycles and subtracts by 5 when the cycle ends. This is handles with a 3-bit counter called a in an always block, which also handles the changes to toggle\_bit. If a is equal to 3, then a is reset to 0, and 5 is subtracted from toggle\_counter. Otherwise, toggle\_counter is incremented by 3. This matches the given waveform.

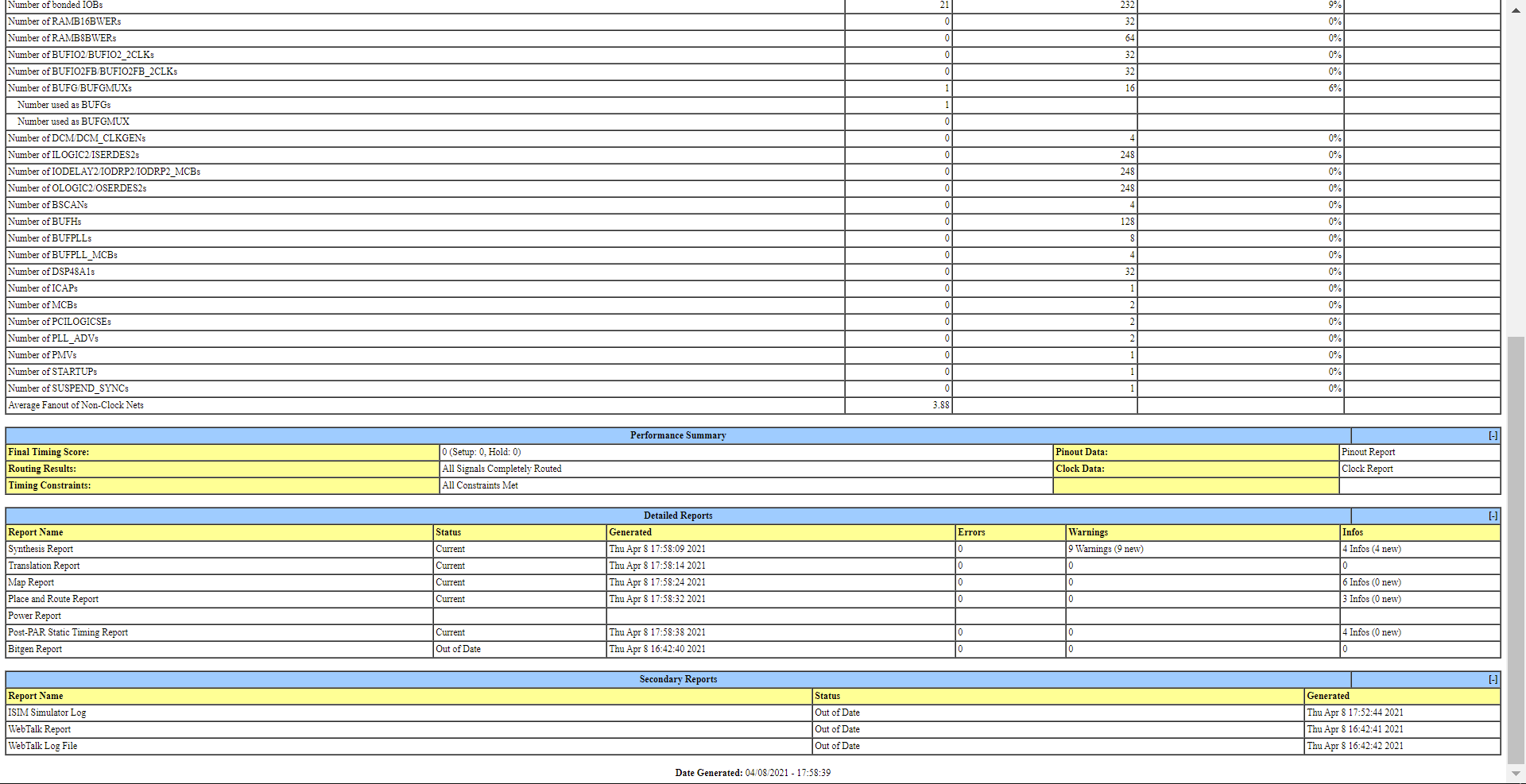
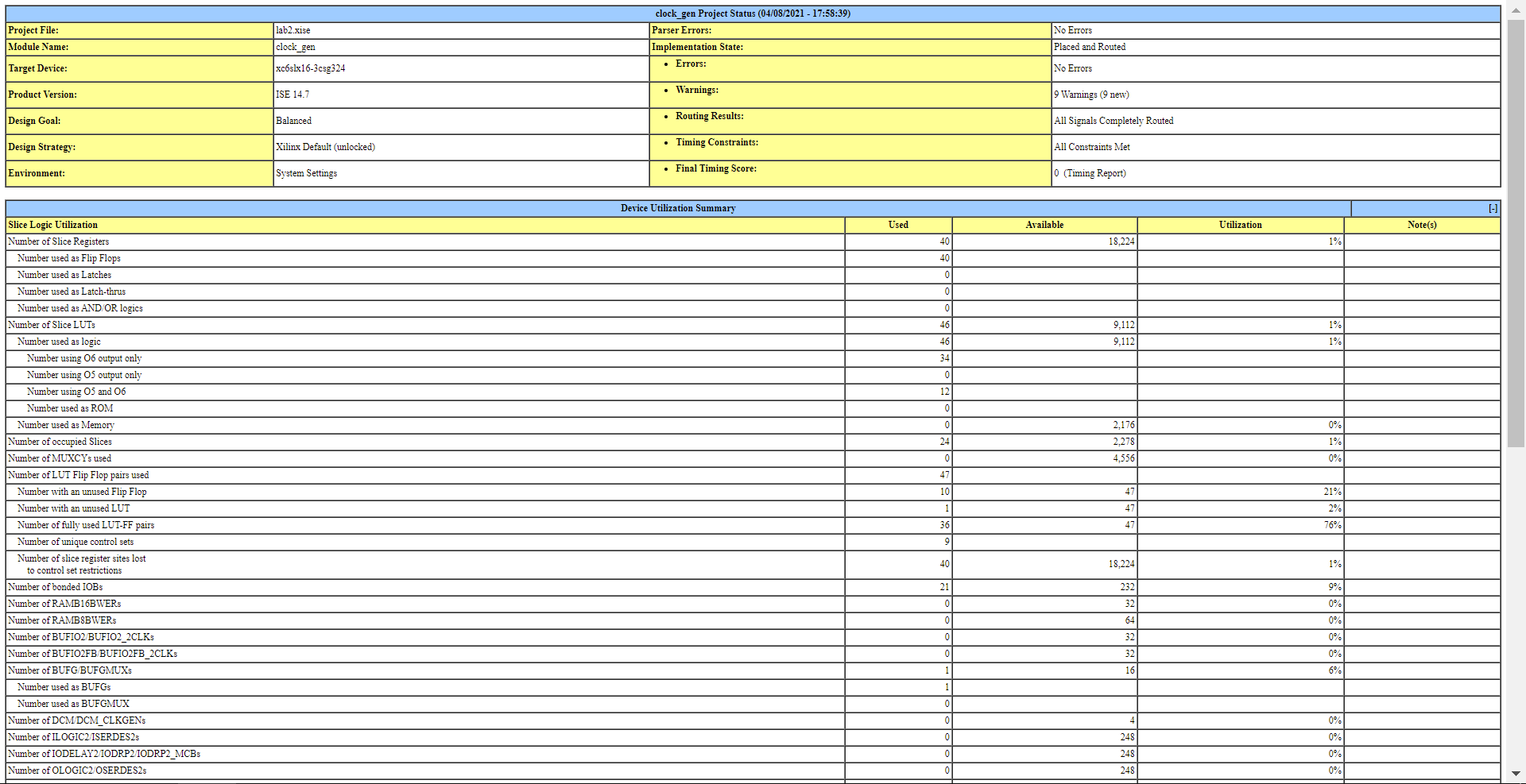
Testbench design:

My testbench tests the module by setting rst to 1, then setting rst to 0 and waiting. This ensures that the output is correct during reset and after. I confirmed that the module was working correctly by examining the waveform output and comparing it to the provided waveforms. My waveforms match the given ones identically, starting at the same point. My waveforms are provided below.

Conclusion:

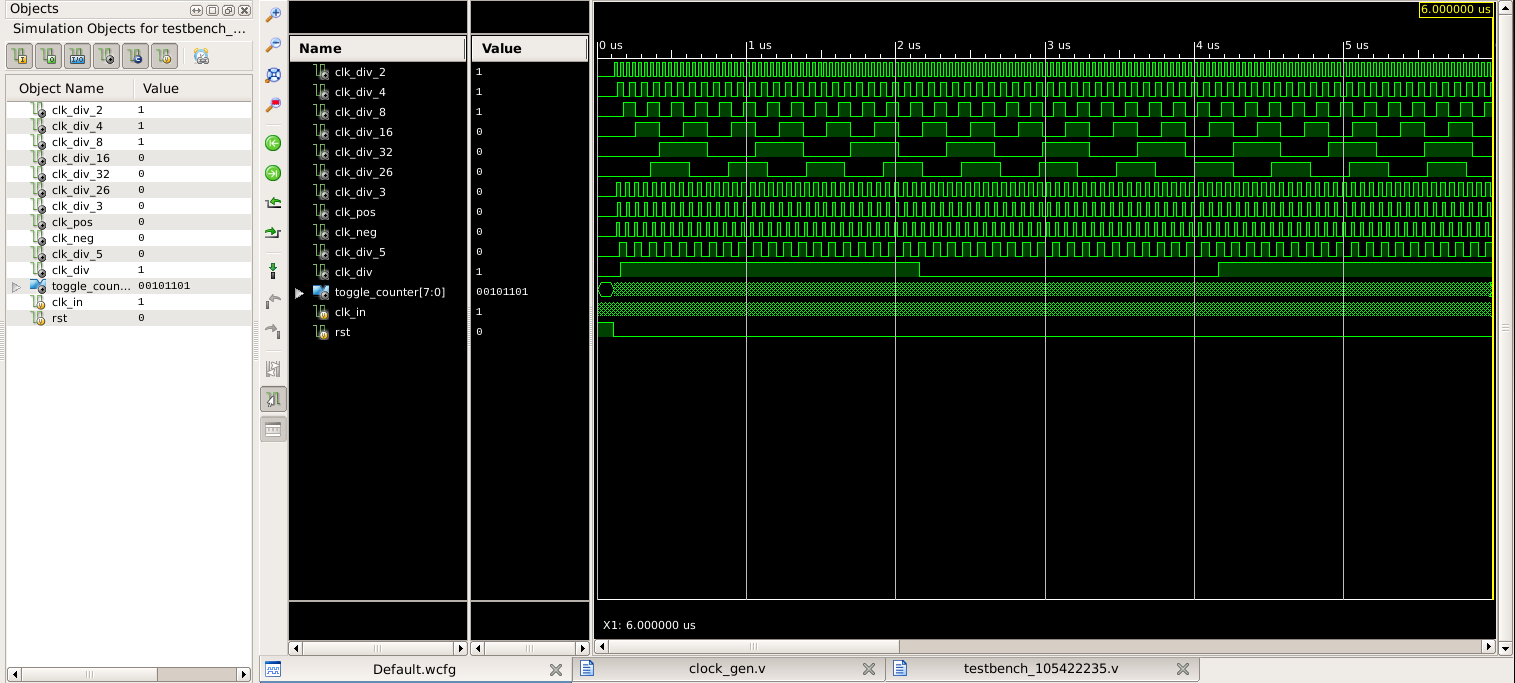
Overall, my implementation uses seven submodules and one top module to handle each task. It outputs twelve variables, as described above. There were multiple difficulties I ran into with the project. I was confused by the implementation of tasks 4, 5, and 6, thinking the output of task 4 was clk\_pos, task 5 was clk\_neg, and task 6 was clk\_div\_3, because of their ordering. I initially implemented this task differently too, directly counting clk\_in instead of using the positive edge or the negative edge. Overall, I enjoyed the project, and found myself learning a lot from the challenges I faced.

ISE Design Overview Summary Report:

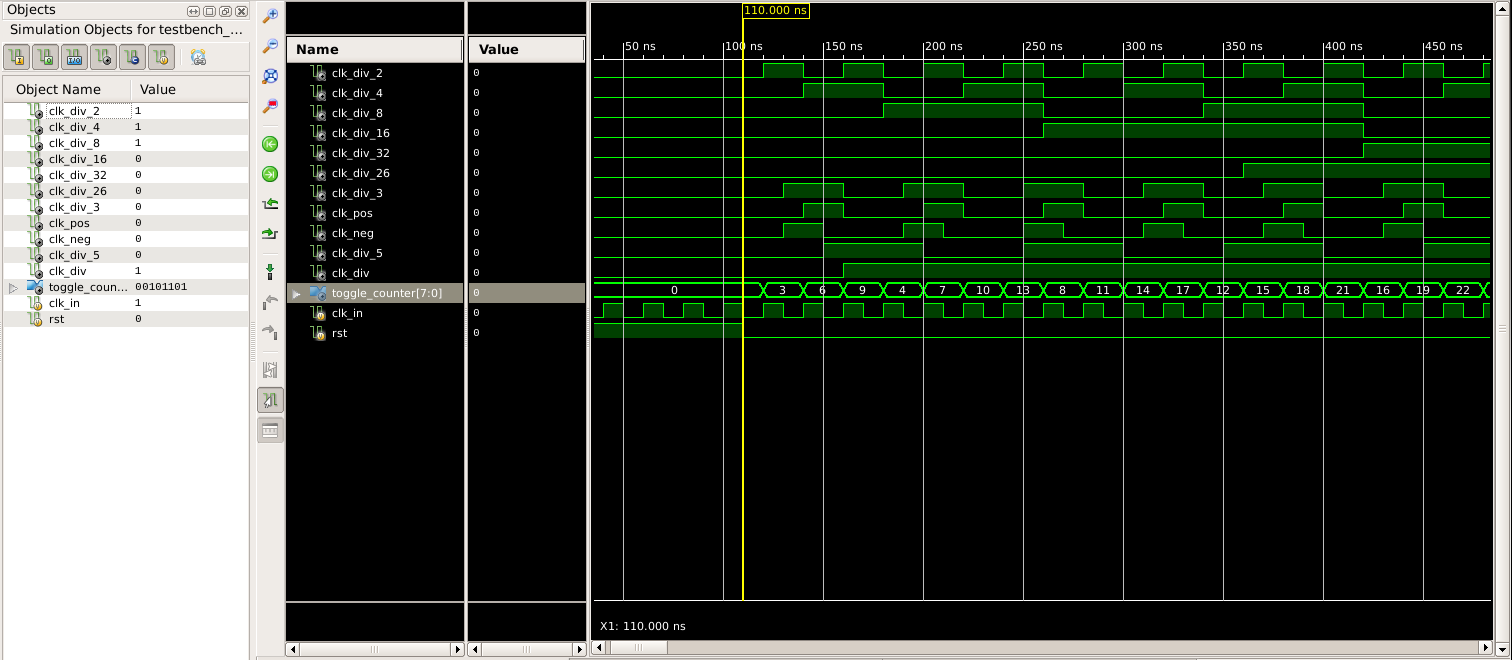


Simulation Output:

Zoomed out:



Zoomed in:



Hand-drawn Design Schematic:

