
Contents

Contents	1
1 Mod 10 Counter	3
1.1 Objective	3
1.2 Discussion	3
1.3 Mod10 Counter	3
1.4 Do file	7
1.5 Testbench	8
1.6 Turn in	8

When clicking on a link in Adobe use alt+arrow left to return to where you started.

Laboratory 1

Mod 10 Counter

1.1 Objective

The objective of this lab is to design a mod10 counter to represent the digits of a stopwatch.

1.2 Discussion

A stopwatch is a device that is used to measure time intervals, usually in competitive events. The stopwatch that you will be designing gets its input from two buttons. The stopwatch will measure down to a 1/10th of a second. The time will be displayed using 3 digits which will represent tenths of a second, unit second and tens of seconds. As a result, the stopwatch is limited to measuring intervals of time from 0.1 second to 99.9 seconds.

Before diving into the architecture of the datapath, you will need to first build an important building block, the mod10 counter.

1.3 Mod10 Counter

A mod 10 counter counts up from 0 to 9 and then rolls over back to 0 to count up again. The term “mod” comes from the word modulus. If you take a number x and form “ $x \bmod 10$ ” you get the integer remainder after division by 10. For example, $12 \bmod 10$ is equal to 2 because $12/10 = 1$ with a remainder of 2. Note that “ $x \bmod 10$ ” will always produce a value between 0 and 9. Thus, a mod 10 counter will count up from 0 to 9 and then back to 0 to start over again.

You will build the mod 10 counter shown in Figure 1.1. The **enb** input enables the counter to count up on a rising edge of the clock. The **synch** input causes the mod10Counter to (synchronously) reset of the rising edge of the clock. The **roll** output indicated when the **currentCount** is going to roll-over from 9 to 0.

Table 1.1 is the truth table for the **currentCount** output. When the **enb** input is at logic 1, **currentCount** is incremented mod 10 on the next positive **clk** edge. When the **synch** input equals 1, **currentCount** goes to 4'b0000 on the next positive **clk** edge. The **synch** input

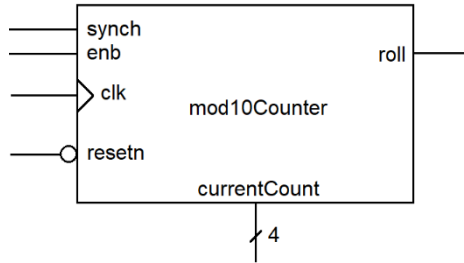


Figure 1.1: The high-level interface for the mod 10 counter.

takes precedence over the **enb** input, so if both are at logic 1 then **currentCount** goes to 4'b0000 on the next positive **clk** edge.

Table 1.1: The truth table for the currentCount output from the mod10Counter.

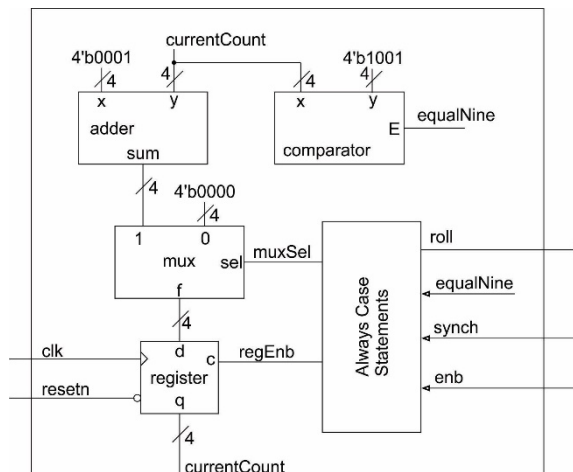
reset	clk	enb	synch	currentCount	Note
0	x	x	x	0	Asynch reset
1	0, 1, ↓	x	x	currentCount	No clk edge
1	↑	0	0	currentCount	Hold
1	↑	0	1	0	Synch reset
1	↑	1	0	(currentCount + 1) mod 10	Count up
1	↑	1	1	0	Synch reset

Table 1.2 is the truth table for the roll output. If **enb** is logic 1 when the **currentCount** equals 9, the **roll** output equals logic 1. In all other cases the **roll** output should equal logic 0. Note, the roll output does not depend on the **clk**, it's a combinational logic block.

Table 1.2: The truth table for the roll output from the mod10Counter.

enb	currentCount	roll
1	currentCount < 9	0
1	currentCount == 9	1

Now that you have a solid grasp of how the mod10Counter should work, let's turn our attention to how this is accomplished. The internal organization of the mod10Counter is shown in Figure 1.2.



You have been provided with the adder, comparator, mux, and register shown in Figure 1.2. In addition to wiring these building blocks together, you will need to define the logic inside the always/case block.

Use the truth tables in Table 1.1 and Table 1.2 along with the hardware organization in Figure 1.2 to fill in Table 1.3 .

Table 1.3: The truth table for the always/case logic inside the mod10counter.

enb	synch	equalNine	muxSel	regEnb	roll
0	0	0			
0	0	1	x		
0	1	0			
0	1	1			
1	0	0			0
1	0	1		1	
1	1	0			
1	1	1			

When you code your always/case statement, with a three-bit output and then have three *assign* statements that break this 3-bit output into individual bits for muxSel, regEnb and roll.

You will need to demonstrate that your mod10counter operates correctly by running the provided testbench. In order for you to verify correct operation, you need to understand what output the simulation should output so that you can compare that to what your simulation actually producing. Any difference between these two indicate an error (either in your understanding or circuit behavior) that need to be fixed. To do this complete the timing diagram in Figure 1.3 using the value from the testbench.

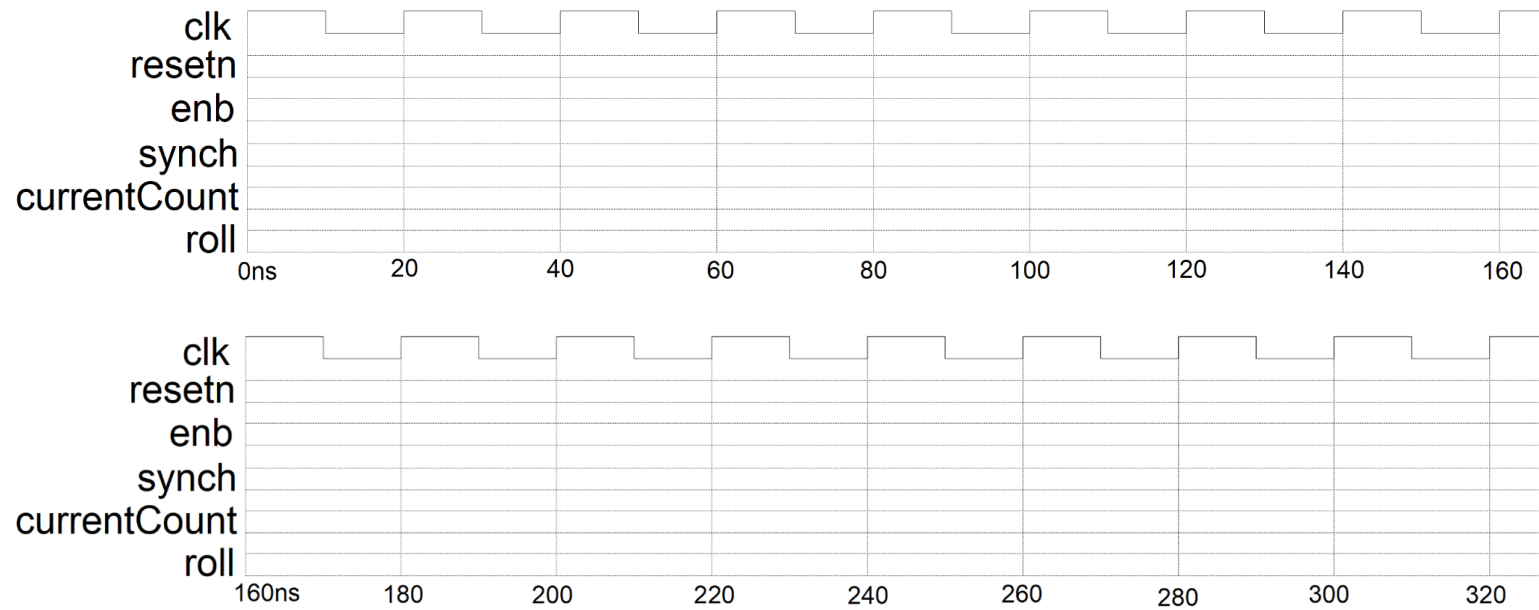


Figure 1.3: A timing diagram for you to fill out based on the testbench code for the mod10Counter. Note the diagram was too wide to be readable in one line, so it has been broken in half at 160ns to make it easier to you to read.

1.4 Do file

You have been required to run simulation for almost every lab. The goal of this requirement is not to just give you an extra task, but rather to expedite your debugging of your Verilog. However, if you have a lot of errors in your design, you may need to run the simulation multiple times. Setting up all the signals, their colors and radix can be a time consuming (and time wasting) task. The solution to this problem is to create a script that sets up all the signals, their order, colors and radices. This script is called a do file.

Listing 1 shows a partial do file for the mod10counter testbench. The “#” symbol is used to denote comments – any text placed after them is ignored by the do file interpreter.

Listing 1.1: A partial do file for the mod10counter.

```
#####
# Search Internet 'modelsim command reference manual'
#####
vsim work.mod10counter_tb
restart -f
delete wave *

add wave -position end sim:/mod10counter_tb/uut/clk
      <<you need more add waves here>>
add wave -position end -radix unsigned -color greenyellow sim:/mod10counter_tb/uut/currentCount
```

The first three lines start the simulation and delete any waveforms that may be left over from a previous simulation. I included this in the do file because I sometimes rerun a simulation to observe the modules behavior at some earlier time.

The waveforms in the simulation are added using the “add wave” command. If you drag-and-drop a signal name into the waveform area, you will see the add wave command appear in the console at the bottom of the ModelSim window. For example when I manually added the clk, I see:

```
add wave -position end sim:/mod10counter_tb/uut/clk
```

I then add the color of the wave and the radix in between “end” and “sim”. See Listing 1 for a couple of examples. You should use notepad to edit the do file. Do NOT use a word processor like MS Word because they tend to change the extensions of files when you save.

After editing the do file, it should be stored in the following folder:

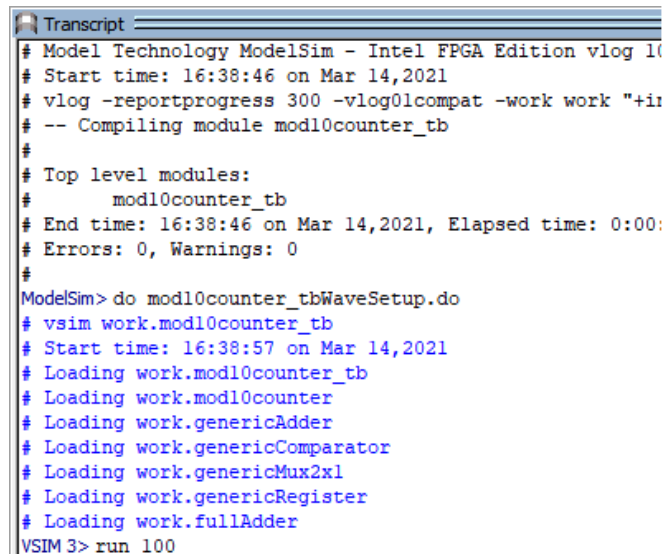
```
<projectDirectory>\mod10Counter\simulation\modelsim
```

Where <projectDirectory> is the path of your project. If this folder does not exist, then you need to have to run “Start Analysis and Elaboration” at least once. You know you are in the project directory when you see the project QPF file. You should run the do file immediately after launching ModelSim from Quartus. You no longer need to open the work directory and right mouse clicking on the testbench file and select “start simulation.” Instead type:

```
do mod10counter_tbWaveSetup.do
```

When you run the do file, you should see something similar to Figure 1.4.

The ModelSim console allows tab completion, a feature that helps you fill-in the characters of long file name. To use tab completion, type the first few characters of a command/filename and then press Tab. If there are no other file names that match what you have typed in, the remainder of the file name will be auto-complete for you. If there is more than one filename choice, the command/filename will be completed up to the ambiguity and the console will provide a list of candidate filenames.



```

# Model Technology ModelSim - Intel FPGA Edition vlog 10
# Start time: 16:38:46 on Mar 14,2021
# vlog -reportprogress 300 -vlog01compat -work work "+i:
# -- Compiling module mod10counter_tb
#
# Top level modules:
#   mod10counter_tb
# End time: 16:38:46 on Mar 14,2021, Elapsed time: 0:00:
# Errors: 0, Warnings: 0
#
ModelSim> do mod10counter_tbWaveSetup.do
# vsim work.mod10counter_tb
# Start time: 16:38:57 on Mar 14,2021
# Loading work.mod10counter_tb
# Loading work.mod10counter
# Loading work.genericAdder
# Loading work.genericComparator
# Loading work.genericMux2x1
# Loading work.genericRegister
# Loading work.fullAdder
VSIM 3> run 100

```

Figure 1.4: The console output when the mod10counter do file is run.

You can issue a variety of commands in the console window. One of my favorite is “run <time>” to simulate some amount of time. I found this VERY handy when debugging my Verilog code.

1.5 Testbench

Edit the do file provided on Canvas to produce the following output.

clk	default	green trace
reset	default	green trace
enb	default	gold trace
synch	default	gold trace
roll	default	yellow trace
currentCount	unsigned	greenyellow trace

You need to look at the values produced by the mod10counter and compare them against the values in Table 1.1. Look for discrepancies starting at time 0 and only advancing the simulation when everything is correct. You will have to transcend the design hierarchy to find the source of your errors. Most of my errors are due to incorrect wiring or modules – wrong names or wrong signal order.

1.6 Turn in

You may work in teams of at most two. Make a record of your response to the items below and turn them in a single copy as your team’s solution on Canvas using the instructions posted there. Include the names of both team members at the top of your solutions. Use complete English sentences to introduce what each of the following listed items (below) is and how it was derived. In addition to this submission, you will be expected to demonstrate your circuit at the beginning of your lab section next week.

Mod10 Counter Verilog

- Complete Table 1.3 and Figure 1.3.
- Verilog code for the body of the mode10counter module (courier 8-point font single spaced), leave out header comments.
- Timing diagram of the mod10counter using do file in the testbench.
- Do file.