Lab 10

7-segment Display with Time-Division Multiplexing

Updated: April 14, 2020
Version: SystemVerilog 2017

10.1 Introduction

Previously, you built 7-segment display drivers that you had to manually switch between the digits. Now that you know about sequential logic, we can use the clock that drives that logic to do the switching for us. The name given to this technique is time-division multiplexing (TDM), which sounds impressive but you've actually been using it already (only with manual switching).

TDM involves sharing a bus (or in general, any communication channel) by two or more signals that alternate in time. One signal gets the channel for a while, then the other signal gets it. (Much like small children sharing a toy, only signals don't cry.) If using only 2 digits, we can simply use the high and low of the clock signal to provide two time periods, also known as clock phases. During the phase when the clock is low, one of the seven-segment display digits is driven. During the other phase, the other digit is driven. This is exactly what we did with the manual switch, and to display the correct value on each digit, we used a multiplexer (MUX) to alternate between the two 4-bit values that should appear on each digit.

You might be wondering why we should do this TDM-thing anyway. You partially answered this question in the first 7-segment lab. If we can make a circuit to handle a 7-segment display, why not just make two of them and be done with it? The reason is that it takes fewer outputs with TDM, namely 12 wires with TDM (4 anodes + 8 seven-segment wires) versus 32 wires without it (4x8 seven-segment wires, assuming anodes are direct to supply). Wires coming out of the FPGA are a major commodity, so TDM is a must. Further, TDM allows you to make one 4-bit to 7-segment display driver, and then use it for all 7-segment displays. This works because the 7-segment display continues to glow for a little while after turned off. It is short, but if you switch rapidly, it will look like they are on at the same time, and you can save almost three quarters of the logic gates. In hardware, we are always trying to reduce gates, as gates require power, create heat, cost money, and take up space.

Lastly, to light up all 4 digits, we need a way to cycle between them. We used a 2-to-4-bit decoder for this task in the previous lab, and we will do so again. But how do we generate the 2-bit signal that drives it from the clock? The answer: a 2-bit counter.

We will also have an issue with how *fast* the clock runs (100 MHz). It's so fast that the LEDs inside the 7-segment don't have time to fully turn on before it moves to the next digit. So how do we slow it down? Another counter!

10.2 Objectives

After completing this lab, you should be able to:

- Recognize synchronous design methodology for regular sequential circuits (not FSM)
- Develop a parameterized counter-timer module
- Implement a clock-driven, 4-digit display using multiple instances of your counter module

10.3 Pre-Lab

1. According to this article, Wikipedia: Frame Rate, at what rate (in Hz = cycles/second) do most people perceive modulated light to be stable?

2. The section below describes the operation of a timer module. After reading it and based on your frequency value from the previous question, compute an appropriate counter width (number of bits) for the 4-digit, 7-seg display (meaning good-looking output with no flickering) assuming the input clock is running at 100 MHz.

10.3.1 Timer operation

A basic timer (or clock-divider) module can be constructed using a counter. If you look at an individual bit of the counter value, it switches back and forth, 0 and 1, just like a clock, but with a frequency that is less than the input clock. Table 10.1 and Figure 10.1 demonstrate this effect using a 4-bit counter. *clkA* = Count[0] and is 1/2 the speed of the input clock. *clkB* = Count[1] and is 1/4 the speed of the input clock. *clkC* = Count[2] and is 1/8 the speed of the input clock. (Remember, clock period is a full off-on cycle.)

This is not the ideal way to do clock division because it introduces some *skew* (phase shift between the clocks), which in this case would be equal to the time delay of the counter. However, this method is very common and will work for our purposes because the two elements that need to be clocked together, the MUX and display digits, are both driven by the output of our timer module. Thus, they will be synchronized.

10.4 Procedure

The only new component here is the counter-timer. After designing it, you will need to create a new mid-level file that is your 4-digit display driver. You will then create a top-level file that connects your ALU from the previous lab to the display driver to create a (very simple) calculator! In this sense, we will be building modules from the bottom up. Thinking top down, it would be helpful to have a schematic (a picture is worth 1000 words), but I want you to create it for yourself. More on this below.

10.4.1 Counter

The Pre-lab section described the operation of a counter-timer. You'll notice that the code in Listing 10.1 is very similar to the register from the previous lab, except for a few differences:

1. There is no Data input. We don't need one. (Unless we wanted to build a counter that you could "load" with a specific value, which can be done.)

2. There is a one bit output `tick`. This is the "timer" part of the counter-timer. We can use it to get a short pulse when the counter reaches its maximum value.

3. A "next-state logic" section is added. This defines what the next value of the internal register (`Q_reg`) should be. Note that this is combinational logic (`@*`) as opposed to the sequential logic of the register `always` block.

4. We moved the enable (`en`) signal to the next-state logic. This is a better practice because it's more tightly grouped by purpose. The memory code is pure memory, and the next-state logic determines what should be the next value stored.

5. (Note that we could have moved the reset as well, but it would no longer be asynchronous. While we wouldn't notice a difference in this lab, this would be a fundamental design change.)

6. The last line might contain syntax that you have not seen. `{N{1'b1}}` is a special concatenate notation. It means replicate the inner value (`1'b1`) N times. So the result is 11111... (N bits).

Sequential logic is typically defined in this manner – with state memory (register), next-state logic, and output logic. (The next-state and output logic can be combined in a single `always` block, but it's usually not as easy to read.)

As usual, test your `counter` module in simulation after creating it. The test bench will be very similar to your test bench for the `register`, with a clock signal that continuously oscillates.

Table 10.1: Counter-timer count sequence

Count	clkC = Count[2]	clkB = Count[1]	clkA = Count[0]
0000	0	0	0
0001	0	0	1
0010	0	1	0
0011	0	1	1
0100	1	0	0
0101	1	0	1
0110	1	1	0
0111	1	1	1
1000	0	0	0
1001	0	0	1
1010	0	1	0
1011	0	1	1
1100	1	0	0
⋮	⋮	⋮	⋮

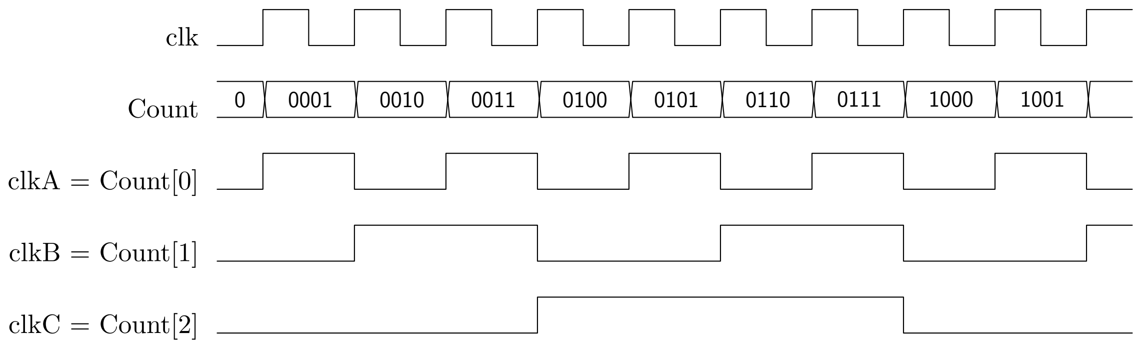

Figure 10.1: Counter-timer waveforms

Listing 10.1: Unfinished counter

```
module counter #(parameter N=1)
   (
   input clk, rst, en,
   output [N-1:0] count,
   output tick
   );

   // internal signals
   reg [N-1:0] Q_reg, Q_next;

   // register (state memory)
   always @(posedge clk, posedge rst)
   begin
      if (rst)
         Q_reg <= 0;
      else
         Q_reg <= Q_next;
   end

   // next-state logic
   always @*
   begin
      if (en)
         Q_next = /*???*/;
      else
         Q_next = Q_reg;   // no change
   end

   // output logic
   assign count = Q_reg;
   assign tick = (Q_reg=={N{1'b1}}) ? 1'
      b1 : 1'b0;

endmodule // counter
```

Listing 10.2: Register from previous lab

```
module register #(parameter N=1)
   (
   input clk, rst, en,
   input [N-1:0] D,
   output reg [N-1:0] Q
   );

   always @(posedge clk, posedge rst)
   begin
      if (rst==1)
         Q <= 0;
      else if (en==1)
         Q <= D;
   end
endmodule // register
```

10.4.2 Mid-level: 7-seg driver

You are now ready to modify your `sseg4` display driver from Lab 8. Create a new file named `sseg4_TDM`. As you do, re-draw the schematic from Lab 8 (Figure 10.2) to include the modifications below. If it gets too messy, get a new piece of paper and re-draw it. This will be one of your deliverables.

1. Remove `digit_sel` from the input list and make it an internal wire (2-bit).
2. Add a 1-bit `reset` input that connects to the reset (`rst`) input on both counters.
3. Add a 1-bit `clock` input.
4. Add a copy of your `counter` module named `timer` that uses the number of bits you determined in the pre-lab. This will be the timer/clock-divider that slows the clock down from 100 MHz to the speed needed by the 4-digit display.
   (a) The `clk` input comes from the new `clock` input.
   (b) The `en` input is a constant 1 (always on).
   (c) The `tick` output goes to your 2-bit counter (below).
   (d) The `count` output is not used.
5. Add a copy of your `counter` module named `counter2` that is 2 bits. This will count from 0 to 3 to cycle through the digits.
   (a) The `clk` input uses the same `clock` signal as above.
   (b) The `en` input comes from the `tick` output of the `timer` module (above).
   (c) The `count` output drives the `digit_sel` signal.
   (d) The `tick` output is not used.
6. We also need to correct one thing from last time: the BCD converter is connected to the wrong MUX input. That is, when `hex_dec` = 0, the display should be in hex (i.e. hex is default). When `hex_dec` = 1, the display should be in decimal (BCD).

7. *Optional:* Add a parameter named `timerN` to your display driver that is input as the parameter for the timer `N` value to control the clock speed. This will allow you to change the speed in the top-level file rather than in your mid-level file.

Test your display driver by creating your own test bench and corresponding ERT. The test bench will be similar to the `counter`, but you will need to wait for *much* longer as you are slowing down the operation (on purpose) with your `timer` module. Remember several things:

- Your `timer` will produce one pulse on the `tick` output for every 2^N clock cycles.
- The pause command (`#N`) waits for the number of cycles specified by the `timescale` directive at the top of your file. If your clock is running with a period of `#10`, then you will need to wait 10×2^N for a single timer pulse.
- By default, the simulation stops at 1000 ns. You will need to click the Run All button to make it simulate the remaining time.

10.4.3 Top-level: Calculator

You're now ready to put your display driver together with your ALU from the previous lab and create a calculator. To keep it simple, we'll reuse `top_lab9` from last time (although it would be better to modify it slightly). Draw this as another schematic (separate from your earlier one).

1. Create a new file named `calc_lab10`.
2. Create an instance of `sseg4_TDM` named `disp_unit` and an instance of `top_lab9` named `calc_unit`.
3. Connect the `led` output of `calc_unit` to the top-level `led` output as before (i.e. LEDs on the board).
4. Also send the upper 8 bits (so `led[15:8]`) to the *lower* 8 bits of the `data` input of `disp_unit`. Make the upper 8 bits of `data` constant zeros. This will show the output register from the ALU (i.e. result of calculation) on the 7-seg display.
5. Connect the buttons, switches, and 7-seg outputs as before. Recall that switch 15 goes to `hex_dec` and switch 14 goes to `sign`.

Implement on your Basys3 board.

10.5 Deliverables

Submit a report containing the following:

1. Verilog source files written as part of this lab (3 files), corresponding test modules and expected results tables (only 2 simulations)
2. Two schematic drawings of your `sseg4_TDM` module and `calc_lab10` module. Make them clear and readable; re-draw if needed.
3. Answers to the following questions:
   (a) What are the three main "groups" of the RTL definition of sequential logic?
   (b) Copy Figure 10.3b onto your own paper (or do it electronically) and draw three boxes around the components that belong to each group. Include your annotated figure in your report.
   (c) If instead of a counter, you wanted to make a shift register that moved the input bits from right to left (low to high). What would you put on the line `Q_next = /*???*/`?

10.6 Generalizing: Sequential Logic

The standard structure used for the counter above (state memory, next-state, and output logic) is called "register-transfer level" (RTL) description. That is, we are describing operations as registers (memory units) with combinational logic between those registers.

The `counter` code implements the circuit shown in Figure 10.3. On each rising edge of the clock, the register stores a value (D input) that is one greater than the currently stored value (Q output). In other words, the value stored increments (increases by one) on each clock edge.

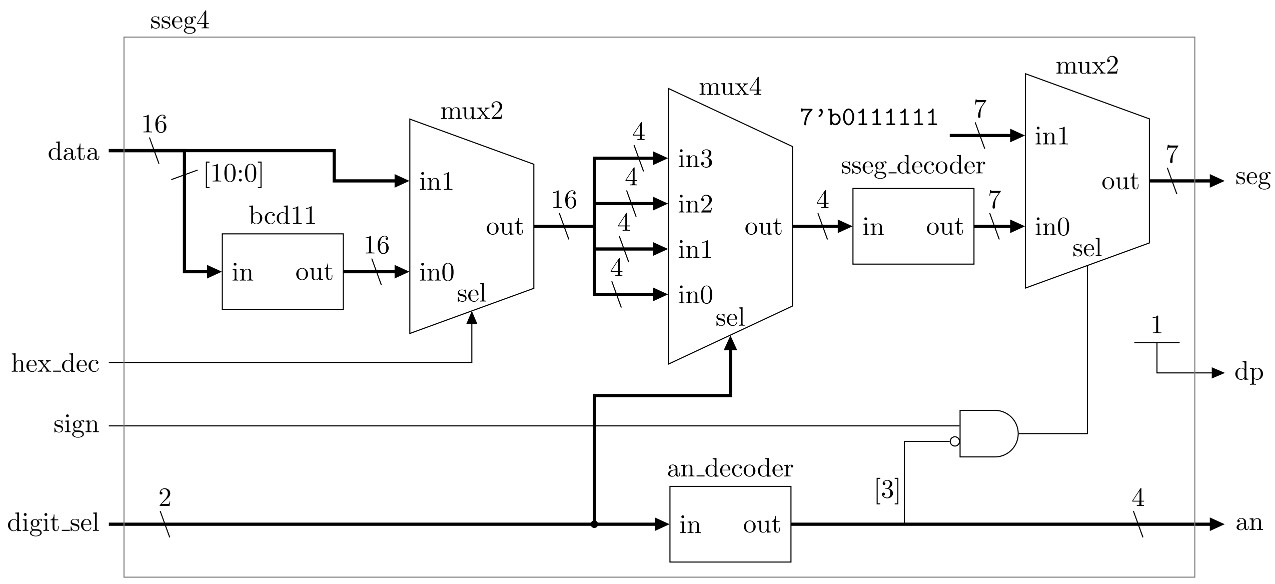

Figure 10.2: 4-digit 7-seg driver (`sseg4`) schematic repeated from Lab 8

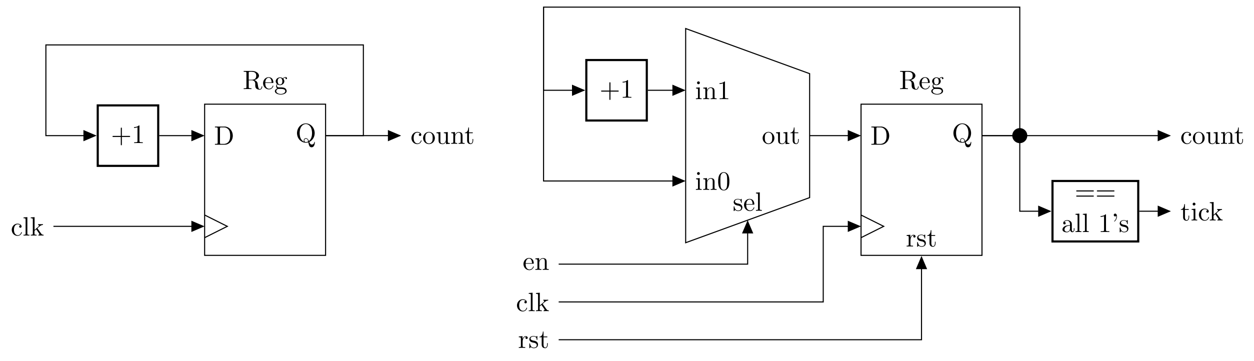

Figure 10.3: RTL schematics for a basic counter and our counter-timer