ECE 351- Spring 2021
Homework #3

Submit your deliverables to your Homework #3 dropbox by 10:00 PM on Monday 24-May-2021 (yes, that is the Memorial Day holiday). The 23 hr, 59 min "grace" period is in place for this assignment but no submissions will be accepted after Noon on Tue, 01-June. We will only grade the submission with the latest date stamp.

Source code should be structured and commented with meaningful variables. Include a header at the top of each file listing the author and a description. You may use the following template:

```
////////////////////////////////////////////////////////////
// <filename>.sv - <one line description>
//
// Author:  <your name> (<your email address>)
// Date:    <date you created the code>
//
// Description:
// ------------
// <text description of what function the module performs>
////////////////////////////////////////////////////////////
```

Files to submit:

- hw3_prob1.sv (serial-in, parallel-out shift register implementation)
- hw3_prob2.sv (your car wash FSM implementation)
- pmodSSD_Interface.sv (your implementation of the PmodSSD interface)
- source code for any starter code files that you changed
- results_hw3_prob3.txt (your transcript from a successful simulation of problem 3)

Note: the name of the SystemVerilog source code files should match the name of the module. The file should have a .sv extension for a SystemVerilog source code file or package. Your transcripts should include your name and email address and the working directory of your simulation per the instructions posted in D2L Announcements.

Create a single .zip or .rar file containing your source code files and transcripts showing that your implementation simulates correctly. Name the file <yourname>_hw3.zip (ex: rkravitz_hw3.zip).

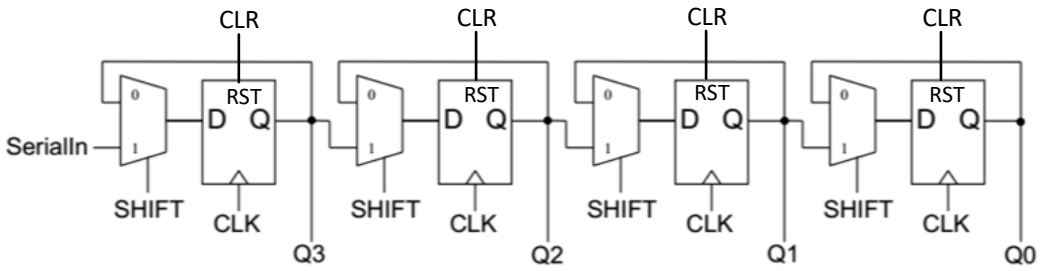
Acknowledgements: The PmodSSD problem (problem 3) was originally given to my ECE 351 class in 2012. I have updated it for SystemVerilog and reworked the assignment a bit. Problems 1 and 2 are based on exercises from Brent Nelson's book *Designing Digital Systems with SystemVerilog (v2.0*)

Problem 1 (15 pts)

In this problem and the next we are going to do some straightforward sequential design. You do not need to simulate your results and/or turn in transcripts. You should, however, structure your code and use meaningful variable names. Since you are not simulating your design, we will grade your code, so correct functioning, neatness, and readability count. You will write this code from scratch – no starter files. There were several comments when I asked about the midterm exam that you did not have enough practice writing SystemVerilog - easy to fix. I am including these problems to help you prepare for the final exam – they are representative of the type of problems you may see on the exam.

The term *shift register* is applied to circuits where flip-flops are wired in series. That is, the output of one flip-flop is wired to the input of the next. All the flip-flops load new values on the clock edge. Since all the flip-flops are edge triggered and all are triggered on the same clock edge, the transfer of these values between flip-flops happens in unison. One common type of shift register is called a serial-in, parallel-out shift registers.

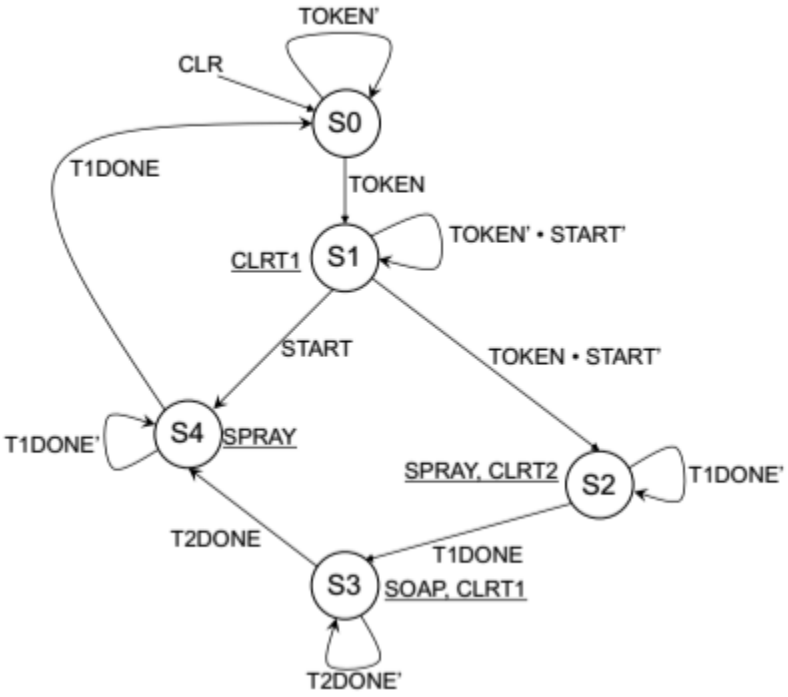
Write a SystemVerilog model for the following schematic: You can assume that all of the flip-flops are clocked on the positive edge of CLK and that when SHIFT = 1'b0 the register loads its old value. When SHIFT = 1'b1 the register shifts right (i.e., each flip-flop loads it left neighbor's value). CLR is a synchronous reset (Q -> 0) for all the flip-flops.



Problem 2 (15 pts)

Understanding how to design and implement Finite State Machines is a must-have skill to be a successful computer engineer. While we normally teach FSM design in digital logic courses, FSM's play just as big a role in many software applications, too.

In this problem we are going to implement a FSM-based controller for a car wash. The car wash supports two kinds of washes. The inexpensive wash (cost = 1 token) simply sprays the vehicle for a set period. The deluxe car wash (cost = 2 tokens) consists of three steps: (1) rinse the vehicle with water, (2) apply soap, and (3) rinse the vehicle again. So, how does the FSM determine which car wash the customer wants? That is what the START signal is for. For an inexpensive car wash the customer inserts a single token and presses the START button. For a deluxe car wash the customer inserts a second token without pressing the START button. A state transition diagram for this controller is shown below:



Design and implement this FSM (a Moore machine since the outputs are only dependent on the current state) in SystemVerilog using the 3 always-block method. Like the previous problem, you do not need to simulate your design, but you do need to write nicely structured, readable code. You may start with the signature on the next page:

```
module carwash_fsm (
   input logic     clk, CLR,    // clock and reset signal.  CLR is asserted high
                                // to reset the FSM
   input logic     TOKEN,       // customer inserted a token.  Asserted high
   input logic     START,       // customer pressed the START button.  Asserted high
   input logic     T1DONE,      // spray time has expired.  Asserted high
   input logic     T2DONE,      // rinse time (after soap) has expired.  Asserted
                                // high

   output logic    CLRT1,       // clear the spray timer.  Assert high
   output logic    CLRT2,       // clear the rinse timer. Assert high
   output logic    SOAP,        // apply soap.  Assert high
   output logic    SPRAY        // turn on the spray.  Assert high
);
```

Problem 2 (70 pts)

Note: The specification for this problem is more complicated than any other problem we have done this term, so take a deep breath, read the documentation, draws pictures, and study the code we provided before you start writing your own code.

This problem consists of completing the design, implementation, and simulation of a single SystemVerilog module that provides an interface to a Digilent PmodSSD two-digit 7-segment display. (http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,401,481&Prod=PMOD-SSD). Yes, we are using SystemVerilog to implement hardware that controls other hardware, in this case a peripheral device.

You have been provided with the PmodSSD datasheet and schematic, a working emulation of a PmodSSD device, a working testbench and a parameterized clock divider. Your task is to implement an interface to the peripheral that has as its input, two character registers (one for each digit), a clock and a reset and provides outputs to drive the control signals on the PmodSSD. Success will be demonstrated by providing a waveform and transcript of your simulation.

Functional Specification

Consider the design of an interface to the Digilent PmodSSD. As you will see from the datasheet, its connections are 7 anode pins (called AA-AG in the datasheet) shared between the two digits, a digit-enable signal (called C in the datasheet) that is switched at a frequency of at least 50Hz to select first one, and then the other digit and VDD and GND. Your interface will need to generate the C and AA-AG signals for the two multiplexed digits, first converting the character codes provided as inputs into the 7 segment values that will control each of the LED segments representing the character. The datasheet provides a good explanation of how to use the digit-enable signal to drive the two digits so no further detail will be provided here.

Deliverables

- Source code for your SystemVerilog module(s) and any others SystemVerilog files that you changed. This would include your testbench that needs to include your name, email address, and simulation working directory.
- A waveform and a console transcript that shows that your implementation works correctly.

Grading for this problem:

You will be graded on the following for this problem:

- Correct implementation of the binary to 7-segment converter function [10 pts]
- Correct implementation of the PmodSSD interface [30 pts]
- The functionality of your design as documented in the waveform and console transcript. [25 pts]
- The quality of your design expressed in your SystemVerilog source code. Your code should be cleanly structured and commented - a joy to read. [5 pts]

Task list

STEP 1: Download the Homework #3 release package

Download and unzip ecw351sp20_hw3_release.zip from D2L. This .zip file contains this document and others, as well as a PmodSSD emulator and testbench. A waveform configuration file (wave.do) for QuestaSim is also included. While this file should work for your design, you will need to modify it if your signal names or hierarchy do not match.

STEP 2: Study/Understand the datasheet, schematic and the HDL provided

You will note that this problem write-up is briefer and provides less detail than the previous write-ups. While this may make you feel uncomfortable, the information you need to complete this project is in the HDL code and the documentation that has been provided. As fledgling SystemVerilog designers (you must admit, you've come a long way since the start of the term) you should be at the point where you can focus less on syntax and tools and more on creating an original design – that's the main objective for this problem.

STEP 3: Implement the character code to 7-segment decoder

In a display subsystem such as this one, the data to be displayed is presented to the interface using a character code. That character code needs to be translated into the individual signals that are driven to each of the segments of a digit (the AA..AG signals on the PmodSSD). For this problem we will use the following translation table:

Character Code	Displays
0 - 9	Characters 0 to 9
10 – 15	Upper case characters A to F
16 – 22	Single segments a to g
23	Space (Blank)
24*	Upper case character H
25*	Upper case character L
26*	Upper case character R
27*	Lower case character L (l)
28*	Lower case character R (r)
29 – 31	Space (blank)

* These special characters are easy to produce on the 7-segment display because they map well to the segments. The exceptions are the Upper Case R which looks just like an upper case A and the lower case L which looks like a 1 (one). The good news is that you can use the same 7-segment patterns for both. The bad news is that your user cannot tell them apart…everything is a tradeoff ***sigh***

There are several ways to implement this functionality. For example, you could place the decoder in a separate module or you could include a case statement in your interface. For this problem, please implement the decoder using a SystemVerilog function in your pmodSSD_Interface code. Functions were discussed in class and there is an example of a similar function in the testbench. A suitable

translation table is provided with the release.

STEP 4: Implement the PmodSSD Interface

The purpose of the PmodSSD interface is three-fold:

First, the digit to be displayed needs to be translated from a character code to the 7-segment anode signals. Include the function you wrote in Step 3 in this module to perform the conversion. The 7 outputs from the conversion function are routed to the LED segments (abcdefg) in a display digit. For the PmodSSD display, an LED segment is lit by driving a logic 1 (1'b1) on the corresponding anode line. So, producing, for example the digit "5" on the display, the 7 anodes for that digit would be driven to 7'b1011011 (segments a, c, d, f, g) lit.

Second, the cathode signal (digit-enable) must be generated. This can be a 50% duty-cycle pulse of any frequency greater than 50 Hz. You can instantiate the clock divider module provided in the release to generate this signal. Assume that the input clock frequency is 100 MHz. The clock divider module generates a pulse that is high for a single 100 MHz clock cycle when the counter reaches its top count. This pulse can be converted to the 50% duty cycle signal to drive the digit-enable (the cathode signal) with the following code:

```
// square up the clock to drive the digit enable
// This will give you a 50% duty cycle clock at 1/2 the frequency
always_ff @(posedge clk) begin: digit_enable
   if (reset)
      tick_60Hz <= 1'b0;
   else if (tick_120Hz)  // 2x the frequency for 50% on / 50% off
      tick_60Hz <= ~tick_60Hz;
   else
      tick_60Hz <= tick_60Hz;
end: digit_enable
```

Third, you must multiplex the 7-segments from each digit onto the shared anode signals. This can be done with a 2:1 multiplexer. The select signal for this multiplexer is the cathode (digit-enable) signal produced from the clock divider.

You can use the following module definition to remain compatible with the provided test bench.

```
module pmodSSD_Interface
#(
   parameter SIMULATE = 1     // set to 1 for this project, else it will take
                              // 100's of thousand cycles of simulation time
                              // for each digit change.
)
(
   input logic         clk, reset,            // clock and reset signals
   input logic[4:0]    digit1, digit0,        // digit character codes
   output logic        SSD_AG, SSD_AF,        // Anode segment drivers
                       SSD_AE, SSD_AD,
                       SSD_AC, SSD_AB, SSD_AA,
```

```
   output logic        SSD_C                   // Common cathode "digit enable"
);
```

STEP 5: Integrate your module with the PmodSSD emulator and testbench

The testbench for this problem instantiates and connects your PmodSSD_Interface module to the PmodSSD emulator (`pmodSSD_emu.v`) provided in the release. The emulator was derived from the PmodSSD schematic and models the PmodSSD accurately. You should have studied this code already (Step 2) but it will greatly help your debug effort if you take the time to understand the code in the module. Of note is that besides de-multiplexing the shared anode signals, the output includes its own translation table (written as a SystemVerilog function) that converts your 7-segment output into ASCII characters that can be displayed by the simulator. In effect, the emulator `dsply_digits` ouput port looks like a 2 digit 7-segment display.

STEP 6: Test your design with the provided test bench

The testbench (`hw3_prob3_tb.v`) runs through all of the character codes, driving digit0 with the character code and digit1 with its inverse. This serves two purposes. First, your 7-segment translation function is fully tested and second, by driving one digit as the inverse of the other the testbench also tests your multiplexing function.

The `/sim` directory in the project release provides a waveform configuration file (`wave.do`) that should prove useful in debugging your design and submitting your results. The `wave.do` file makes use of the ability for QuestaSim to create new display variables (combinations of existing variables) and display waveform values in Radix ASCII.

References

[1] Digilent PmodSSD™ Peripheral Module Board Reference Manual

[2] Digilent Seven Segment Display Module Schematic

[3] Wikipedia articles on 7-segment displays