CSC258 - Lab 5

Clocks and Counters

Summer 2018

1 Learning Objectives

The purpose of this lab is to learn how to create counters and to be able to control when operations occur when the clock rate is too fast to be controlled manually.

2 Marking Scheme

This lab, like other labs, worth 4% of your final grade, but you will be graded out of 8 marks for this lab, as follows:

- Prelab - Simulations: 3 marks
- Part I (in-lab): 1 mark
- Part II (in-lab): 2 mark
- Part III (in-lab): 2 marks

3 Preparation Before the Lab

You are required to complete Parts I to III of the lab by writing and testing Verilog code. Include your schematics, Verilog, and simulations outputs (*i.e.* screenshots of simulation outputs) for Parts I to III in your prelab. You must simulate your circuit with ModelSim (using reasonable test vectors you can justify). You should also answer the questions in the handout that are marked as **(PRELAB)**.

In-lab Work

You are required to implement and test all of Parts I to III of the lab. You need to demonstrate all parts to the teaching assistants.

4 Part I

Consider the circuit in Figure 1. It is a 4-bit synchronous counter that uses four T-type flip-flops. The counter increments its value on each positive edge of the clock if the *Enable* signal is 1 (*i.e.* high). The counter is reset to 0 by setting the *Clear_b* signal low, which means that the clear is an **active-low asynchronous** clear. You should implement an 8-bit version of this counter.

An **asynchronous clear** means that as soon as the *Clear_b* signal changes (here from 1 to 0 since we have an *active-low* signal), irrespective of whether this change happened at the positive clock edge or not, the T flip-flop should be reset. This is contrary to the **synchronous reset**, which you implemented in the previous lab, where the D flip-flop could be reset only at the positive edge of the clock.

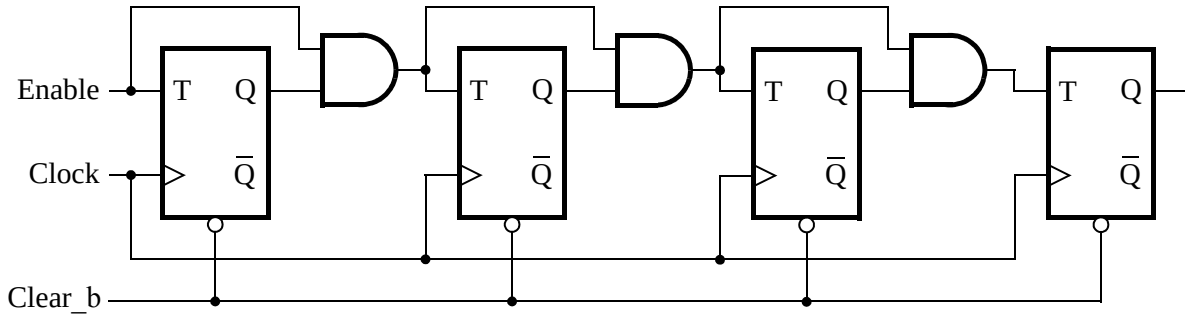

Figure 1: A 4-bit counter.

HINT: Since the state of the flip-flop can change both at the positive edge of the clock or asynchronously when the *Clear_b* signal becomes low, you need to include both signals in the sensitivity list of your always block. You can separate multiple signals in the sensitivity list with commas as follows, while adding a *posedge* or *negedge* transition keyword as needed:

always @(<**edge**> signal_a , <**edge**> signal_b)

Older Verilog standards, which are still supported, used the word *or* in the sensitivity list instead of a comma. Note that `signal_a` has to be different than `signal_b`.

For this part, perform the following steps:

1. Draw the schematic for an 8-bit counter using the same structure as shown in Figure 1. **(PRELAB)**
2. Annotate all Q outputs of your schematic with the bit of the counter ($Q_7Q_6Q_5Q_4Q_3Q_2Q_1Q_0$) that they correspond to. **(PRELAB)**
3. Write the Verilog corresponding to your schematic. Your code should use a module for the flip-flop that is instantiated eight times to create the counter. Note that you should not name your module `TFF` as this is a Quartus primitive and thus Quartus will ignore that entity after issuing a warning (*e.g.* use a name such as MyTFF). **(PRELAB)**

   TIP: You should name your inputs and outputs in a way that makes it easy to interpret your simulations (*e.g.* use input/output names from your schematic). You can later enclose your module inside of another module that will connect inputs and outputs to SW, KEY, $HEX0$ and $HEX1$ so that Quartus can correctly assign them to appropriate FPGA pins.
4. Simulate your circuit in ModelSim to verify its correctness. You will need to reset (clear) all your flip-flops early in your simulation, so your circuit is in a known state. Include screenshots of simulation output in your prelab. **(PRELAB)**
5. Augment your Verilog code to use the push button KEY_0 as the *Clock* input, switches SW_1 and SW_0 as *Enable* and *Clear_b* inputs, and 7-segment displays *HEX0* and *HEX1* to display the hexadecimal value of your counter as your circuit operates. Simulate your circuit to ensure that you have done this correctly (inlucde at least one screenshot). **(PRELAB)**
6. Create a new Quartus Prime project for your circuit. Make sure it is stored in your `W:\` drive. Do not forget to select the correct FPGA device (5CSEMA5F31C6) and import the pin assignments. Compile the circuit in Quartus and answer the following questions: **(PRELAB)**

   (a) What percentage of FPGA logic resources are used to implement your circuit? **(PRELAB)**

   In Quartus, look at your `Logic Utilization (in ALMs)` in the Fitter Report, which can be located in Quartus `Tasks` pane, usually situated on the left side of the screen. After compiling the design, expand (by clicking on the little `+` sign) `Compile Design -> Fitter`, then double-click on `View Report`. ALM stands for Adaptive Logic Module, which is a basic building block of the FPGA device. Logic utilization is an indication of how many FPGA

resources are used to build your circuit. How does the size of your circuit compare to the size of the FPGA you are using? You can also observe how many registers your design uses. In Quartus terminology, one register is equivalent to one flip-flop. **(PRELAB)**

(b) What is the maximum clock frequency, F_{max}, at which your circuit can be operated? **(PRELAB)**

To find the maximum frequency, compile your design in Quartus, and then run the TimeQuest Timing Analyzer. To do this, in the Tasks pane on the left expand `TimeQuest Timing Analysis` then underneath this menu double-click on `TimeQuest Timing Analyzer`. A new window will open with its own Tasks pane. Scroll down in its Tasks pane and expand `Reports -> Datasheet`, then double-click on `Report Fmax Summary`. The tool reports two maximum frequencies (F_{max}). The smaller reported frequency is the maximum frequency that your design could run at on the DE1-Soc board FPGA.

7. Use the Quartus Prime RTL Viewer to see how the Quartus Prime software synthesized your circuit. You can access the RTL viewer on Quartus via `Tools -> Netlist Viewers -> RTL Viewer`. You can zoom into the various building blocks of your circuit by double-clicking on them, to get more information about their implementation. What are the differences in comparison with Figure 1? **(PRELAB)**

8. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit. When you are sure that it is working correctly, demonstrate it to TAs. **(IN-LAB)**

5 Part II

Another way to specify a counter is by using a register and adding 1 to its value at each iteration. This can be accomplished using the following Verilog statement:

```
Q <= Q + 1'b1;
```

Figure 2 is a code snippet a counter that counts from 0 to F in hexadecimal. The counter also has an active-low synchronous clear (*reset_n*), a parallel load feature (*par_load*), and an enable input (*enable*) to turn the counting on and off.

```
wire [3:0] d;                    // Declare d
wire clock;                      // Declare clock
wire reset_n;                    // Declare reset_n
wire par_load, enable;           // Declare par_load and enable
reg [3:0] q;                     // Declare q

always @(posedge clock)          // Triggered every time clock rises
begin
    if (reset_n == 1'b0)         // When reset_n is 0
        q <= 0;                  // Set q to 0
    else if (par_load == 1'b1)   // Check if parallel load
        q <= d;                  // Load d
    else if (enable == 1'b1)     // Increment q only when enable is 1
        begin
            if (q == 4'b1111)    // When q is the maximum value for the counter
                q <= 0;          // Reset q into 0
            else                 // When q is not the maximum value
                q <= q + 1'b1;   // Increment q
        end
end
```

Figure 2: Example counter code fragment

In this implementation, q is declared as a 4-bit value, which makes this a 4-bit counter. The check for the maximum value is not necessary in the example above. Why? **(PRELAB)**If you wanted this 4-bit counter to count from 0-9, what would you change? **(PRELAB)**

In this part of the lab you will design and implement a circuit using counters that successively flashes the hexadecimal digits 0 through F on the 7-segment display *HEX0*. You will use two switches, SW_1 and SW_0, to determine the speed of flashing according to the following table:

SW[1]	SW[0]	Speed
0	0	Full (50 MHz)
0	1	1 Hz
1	0	0.5 Hz
1	1	0.25 Hz

Full speed means that the display flashes at the rate of the 50 MHz clock provided on the DE1-SoC board. At this speed, what do you expect to see on the display? **(PRELAB)**(HINT: compute the period of that clock.)

You must design a fully synchronous circuit, which means that every flip flop in your circuit should be clocked by the same 50 MHz clock signal.

To derive the slower flashing rates you should use a special kind of counter, that we will call it **Rate-Divider**, which is running with the MHz clock. The output of RateDivider will be a slower alternating signal that can be used as the enable signal of another module. Every time RateDivider has counted the appropriate number of clock edges, a pulse will be generated for one clock cycle. Figure 3 shows a timing diagram for a 1 Hz Enable/pulse signal with respect to a 50 MHz clock. This pulse signal will be used to control the enable signal of your main 0 to F counter. How large a counter is required to count 50 million clock cycles? **(PRELAB)**How many bits would you need to represent such a value? **(PRELAB)**

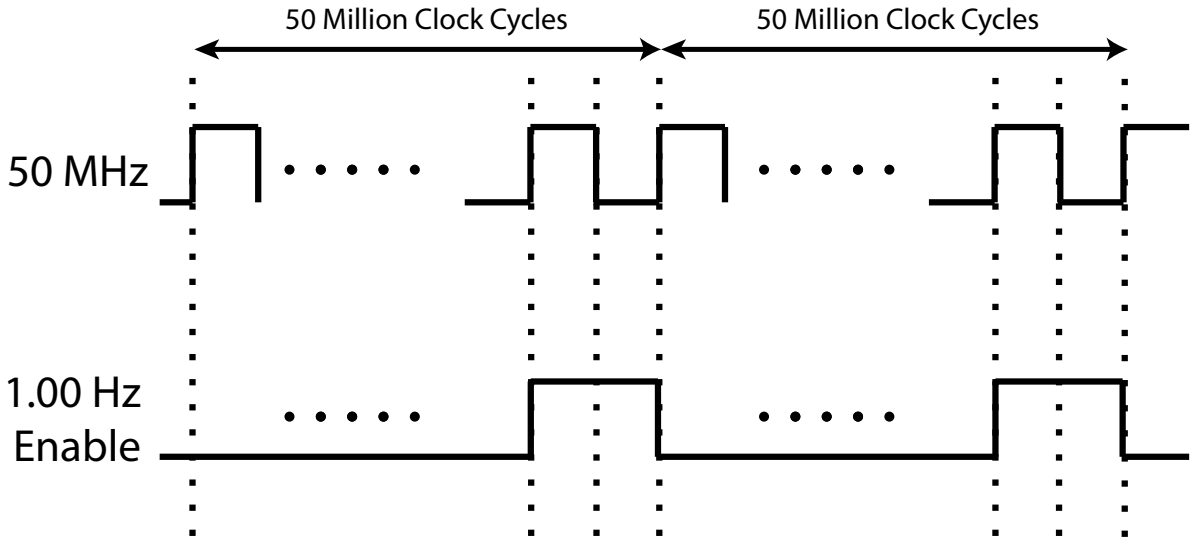

Figure 3: Timing diagram for a 1 Hz enable signal

A common way to provide the ability to change the number of pulses counted is to parallel load the counter with the appropriate value and count down to zero. For example, if you want to count 50 million clock cycles, load the counter with 50 million - 1 (49,999,999). (Why subtract 1?) Then subtract one from it until it reaches 0, and after that reload 49,999,999. Outputting the pulse when the counter is zero can be done using a *conditional assign statement* like:

```
assign enable = (RateDivider == 4'b0000) ? 1 : 0;
```

Note that the above example assumes that RateDivider is a four-bit counter. You will need to adjust this depending on the counter width you use.

These pulses can be used to drive the *enable* signal of the hexadecimal counter (which we will call it **DisplayCounter**) that is counting from 0 through F. Recall that an *enable* signal determines whether a flip flop, register, or counter will change on an active edge of the clock or not.

In summary, you will need two counters. RateDivider will need the ability to parallel load the appropriate value selected by the switches so that Enable pulses are generated at the required frequency. DisplayCounter counts through the hexadecimal values, but only increments when its *enable* input is 1. You may use the sample counter code fragment in Figure 2 as a model to build your counters, adding or deleting features to meet the requirements for each counter.

For this part, you should perform the following steps:

1. Draw a schematic of the circuit you wish to build. Work through the circuit manually to ensure that it will work according to your understanding. **(PRELAB)**

2. Write a Verilog module that realizes the behaviour described in your schematic. Your circuit should have a clock input and two switches inputs. **(PRELAB)**

   HINT: You should name your inputs and outputs in a way that makes it easy to interpret your simulations (*e.g.* use input/output names from your schematic). You can later enclose your module inside of another module that will connect inputs and outputs to SW, and $HEX0$ so that Quartus can correctly assign them to appropriate FPGA pins.

   In addition to switches SW_{1-0} that will be used to control the rate that hex digits are flashed on HEX0, you will also need to use one or two more switches (e.g., as a clear signal). Make sure to label which switches you use for which purpose on your schematic. **(PRELAB)**

   The 50 MHz clock is generated on the DE1-SoC board and available to you on a pin labeled in the *qsf* file as `CLOCK_50`. This means that you can access the 50 MHz clock by declaring an input port called `CLOCK_50` in your top-level module. See Section 3.5 in the DE1-SoC User Manual to learn more about the clocks on the board.

3. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct. You must include screenshots of simulation output in the prelab. You will also need to think about how to simulate this kind of circuit. For example, how many 50 MHz clock pulses will you need to simulate to show that the RateDivider is properly outputting a 1 Hz pulse? **(PRELAB)**

   (HINT: Sometimes simulating a circuit in its entirety is infeasible. It is therefore important to identify which are the critical aspects of the circuit you need to simulate to be confident that your circuit will work. Demonstrating this for a smaller problem can be one valid approach.)

4. Create a new Quartus Prime project for your circuit. Make sure it is stored in your `W:\` drive. Do not forget to select the correct FPGA device (5CSEMA5F31C6) and import the pin assignments. Compile the project. **(IN-LAB)**

5. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit, and demonstrate it to TA when you finished the testing. **(IN-LAB)**

6 Part III

In this part of the lab you are going to *design & implement* a Morse code encoder.

Morse code uses patterns of short and long pulses to represent a message. Each letter is represented as a sequence of dots (a short pulse), and dashes (a long pulse). For example, the last 8 letters of the English alphabet have the following representation:

S • • •
T —
U • • —
V • • • —
W • — —
X — • • —
Y — • — —
Z — — • •

Your circuit should take as input one of the eight letters of the alphabet starting from S (as in the table above) and display the Morse code for it on $LEDR_0$. Use switches SW_{2-0} and push buttons KEY_{1-0} as inputs. When a user presses KEY_1, the circuit should display the Morse code for a letter specified by SW_{2-0} (000 for S, 001 for T, etc.), using 0.5-second pulses to represent dots, and 1.5-second pulses to represent dashes. The time between pulses is 0.5 seconds. Push button KEY_0 should function as an asynchronous reset.

You will likely use a lookup table (LUT) to store the Morse codes, a shift register, and a rate divider similar to what you used in Part II. Let us first look into how we will store the Morse code representation for each letter. Since all the times for dot, dash and pause are multiples of 0.5 seconds, we can use this to our advantage, and represent each Morse code as a sequence of bits. Each bit will correspond to a display duration of 0.5 seconds. Therefore a single 1 bit will correspond to a dot (the LED should stay on for 0.5 seconds), while three 1s in a row (*i.e.*, 111) will correspond to a dash (the LED should stay on for $3 * 0.5$ seconds). In order to differentiate between a dot and a dash, or between multiple successive dots or multiple successive dashes, we will inject zeros between them (*i.e.*, the LED should stay off for 0.5 seconds – the time required between pulses). An LED that is off signifies either a pause (e.g., a transition between a Morse dash and a dot), the end of a transmission, or no transmission.

Using this representation, the Morse code for letter X would be stored as `1110101011100000`, assuming we use 16-bits to represent it. Write the Morse code binary representation of all letters specified above (S to Z) in Table 1 following the same approach. You will observe that a different number of bits is needed for each letter. You will should figure out the maximum number of bits needed when accounting for all letters (S to Z), since your circuit will be much simpler if all letters are stored using the same pattern length. For letters that do not require the maximum number of bits, simply set the last few bits to 0. The last bit of any Morse code representation should be 0.

Fill in Table 1 below as part of your prelab. You will need to decide on the pattern length. Complete the pattern representation for letter X with as many zeros as needed based on the pattern length you chose. **(PRELAB)**

Letter	**Morse Code**	**Pattern Representation (pattern length is _____ bits)**
S	• • •	
T	—	
U	• • —	
V	• • • —	
W	• — —	
X	— • • —	111010101110
Y	— • — —	
Z	— — • •	

Table 1: Morse Pattern Representation with fixed bit-width **(PRELAB)**

The LUT which will store the Morse code patterns (binary representations) can be implemented as a multiplexer with hard-coded inputs corresponding to the required patterns. The output pattern is selected according to the letter to be displayed.

Now that we have stored all possible patterns and can retrieve the pattern that we need, and use it to

display the pattern on the LED, one bit at a time (0.5 seconds per bit). To do that, you need to load the pattern into a shift register (with parallel load). Then, you need to shift the pattern out of the register, one bit at a time, and display each bit on the `LEDR[0]` for the appropriate interval (0.5 seconds per bit). A high-level block diagram of the circuit you are building is shown in Figure 4. Note that various details/connections are not shown there.

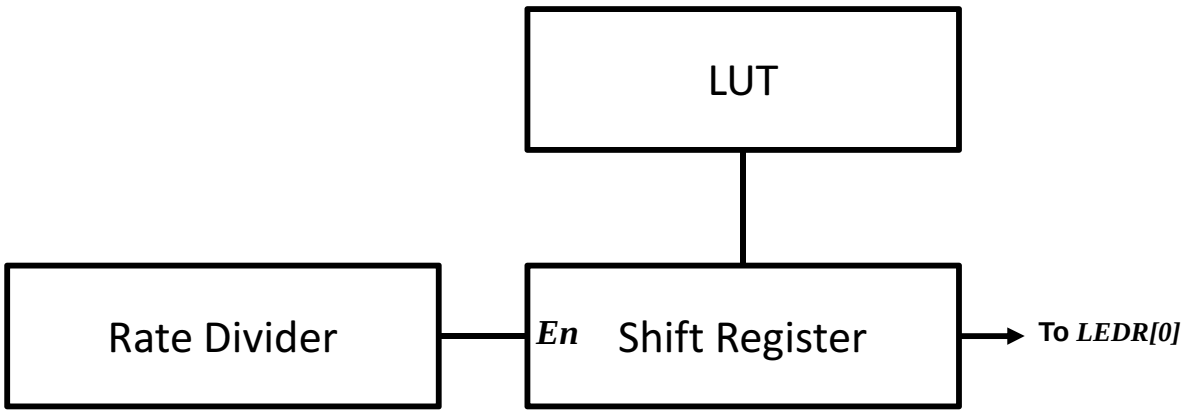

Figure 4: Block diagram of the Morse code circuit

To summarize, once the user presses `KEY[1]` you need to load a shift register with the appropriate pattern (for the letter specified by `SW[2:0]`) and display the Morse code from that pattern on `LEDR[0]`. Note that you should not display a given letter in a loop (*i.e.*, you should re-display the same letter only if the user presses `KEY[1]` again).

The following table summarizes the inputs and outputs that you will use:

Input/Output	Purpose
SW[2:0]	Choose one of the 8 letters S to Z
KEY[1]	Start diplaying Morse code for chosen letter
KEY[0]	Asynchronous reset
LEDR[0]	Output used to display Morse code

In the event that you have not gotten part II of the lab working (*i.e.* you have not been able to implement the 0.5 second enable signal), manually clock your shift register using one of the `KEY` inputs. This will help you to get partial marks.

Perform the following steps:

1. Use Table 1 to determine your codes and bit-width. **(PRELAB)**
2. Design your circuit by first drawing a schematic of the circuit. Think and work through your schematic to make sure that it will work according to your understanding. You can use Figure 4 as your starting point. You should include the schematic in your prelab. **(PRELAB)**
3. Write a Verilog module that realizes the behaviour described in your schematic. You should also inlucde the Verilog code as part of your prelab. **(PRELAB)**

   HINT: You should name your inputs and outputs in a way that makes it easy to interpret your simulations (*e.g.* use input/output names from your schematic). You can later enclose your module inside of another module that will connect inputs and outputs to SW, KEY, and $LEDR$ so that Quartus can correctly assign them to appropriate FPGA pins.
4. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct. If you cannot compile your design to simulate it, then you should at the very least (a) decide what are the inputs settings you want to model and (b) draw waveforms of what you *expect* your output signals to be for those inputs.

You are strongly encouraged to complete this step **before** coming to the lab. However, simulations for this part will not be marked as part of your prelab. They will be marked as part of your in-lab work. **(IN-LAB)**

5. Create a new Quartus Prime project for your circuit. Make sure it is stored in your `W:\` drive. Also make sure that you selected the correct FPGA device (5CSEMA5F31C6) and imported the pin assignments. Compile the project.

6. Download the compiled circuit into the FPGA. Test the functionality of the circuit, and demonstrate it to the TA when it works correctly. **(IN-LAB)**