CSC258 - Lab 5

Clocks and Counters

1 Learning Objectives

The purpose of this lab is to learn how to create counters and to be able to control when operations occur at high clock speeds.

2 Marking Scheme

This lab, like other labs, is worth 6% of your final grade, but you will be graded out of 8 marks for this lab, as follows:

- Prelab: 3 marks
- Part I: 1 mark
- Part II: 2 mark
- Part III: 2 marks

3 Preparation Before the Lab

You are required to complete Parts I to III of the lab by building and testing circuits in Logisim. Include your schematics and Logisim files for Parts I to III in your prelab. You must simulate your circuit inside Logisim, but since the Logisim test vectors only handle combinational circuits, your prelab report will include a test plan and important test cases (which will be tested in your demo). You should also answer the questions in the handout that are marked as **(PRELAB)**.

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. (In Logisim, all the input signals are asynchronous active high by default.) You should implement an 8-bit version of this counter.
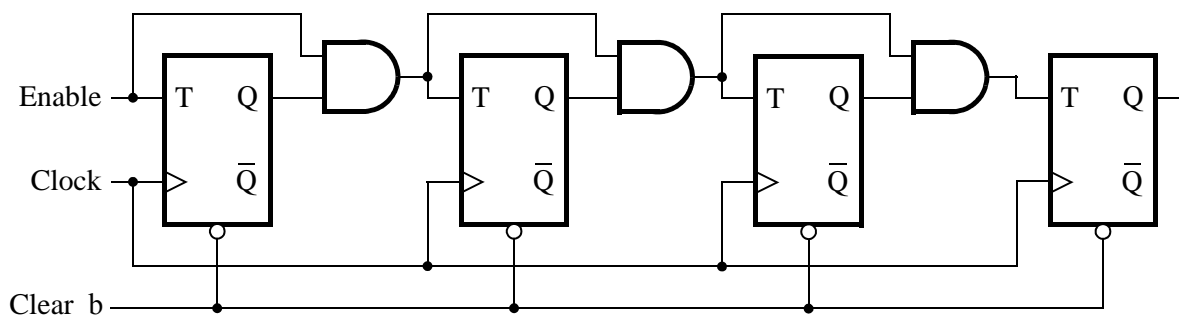

Figure 1: A 4-bit counter.

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 in contrast to the **synchronous reset**, where the D flip-flop can only be reset at the positive edge of the clock.

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 ($Q_7 Q_6 Q_5 Q_4 Q_3 Q_2 Q_1 Q_0$) that they correspond to (where Q_7 is the most significant bit and Q_0 is the least significant bit. Label the inputs according to the diagram in Figure 1. **(PRELAB)**

3. Build the circuit corresponding to your schematic. Your circuit should use the flip-flop module provided by Logisim, instantiated eight times to create the counter. **(PRELAB)**

   Note that for the modules that you are instantiating, it is best if you use the default input and output from the toolbar. It is recommended that 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).

4. Connect the Q output bits to two seven-segment displays so you can monitor the output values. **(PRELAB)**

5. Test your modules with *Poke(* 👆 *)* to verify its correctness. You will need to reset (clear) all your flip-flops early in your simulation, to ensure that your circuit starts in a known state. Include screenshots of simulation output in your prelab. **(PRELAB)**

5 Part II

Another way to specify an counter is by instantiating a register and adding 1 to its value at each iteration. Adding 1 can be accomplished using the *Adder* under *Arithmetic*. The constant value 1 can be found in *Wiring > Constant*, and you can set the value and the number of data bits for this constant in *Properties* after you click on it.
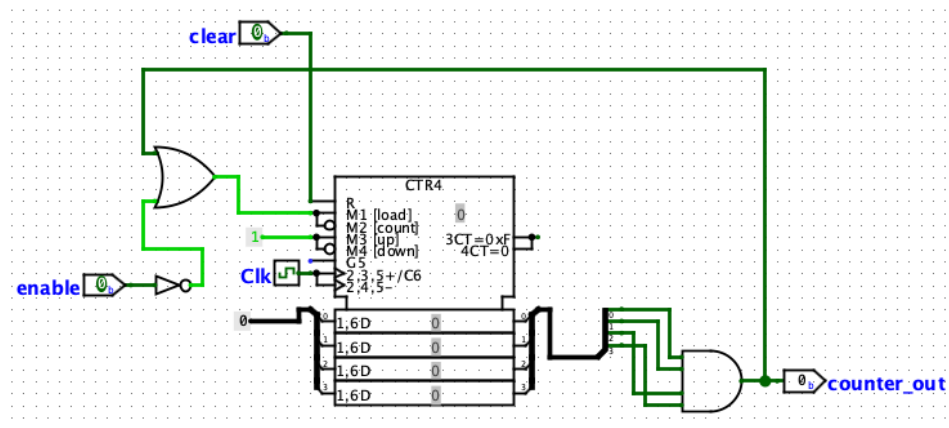

Figure 2: Counter in Logisim

Figure 2 shows an example of a counter in Logisim that counts in hexadecimal from 0 to F. You can find this counter in *Memory > Counter*.

The counter has the following input signals:

1. An active-high asynchronous clear (R at the top left corner),

2. Signals *M1 [load]* and *M2 [count]* that perform load and counter enable, where an input of 1 means loading the value and 0 means start counting

3. Signals *M3 [up]* and *M4 [down]* (note that they are just one signal) which determine whether to count up or down.

4. Inputs called *2,3,5+/C6* and *2,3,5+/C6* that should be connected to a Clock signal. The value that to be loaded needs to be connected the bottom left side of the counter and the value within the counter that is incremented/decremented will be on the bottom right side of the counter.

The implementation above is a 4-bit counter, however you can change this in *Properties* if you wish to use more bits. In your prelab report, provide the answers to the following questions:

1. The check for the maximum value is not necessary in the example above. Explain why in your prelab report **(PRELAB)**

2. If you wanted this 4-bit counter to count from 0-9, how would you adjust the circuit above? **(PRELAB)**

3. In *Properties* there is a setting called *Action On Overflow*. Explain how each value for this setting responds to overflow by experimenting with this setting and describing the results. **(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 a hex display. A two-bit input value to your circuit will be used to determine the speed of flashing according to the following table:

SW[1]	SW[0]	Speed
0	0	Full (16 Hz)
0	1	1 Hz
1	0	0.5 Hz
1	1	0.25 Hz

You must design a fully synchronous circuit, which means that every flip flop in your circuit should be clocked by Logisim's default **16Hz** clock signal. If we were operating with the DE1-SOC boards, the built-in clock would run much faster, at 50 MHz. Logisim provides a slower set of clocks, and we'll use the **16Hz** during simulation.

While you are able to change the rate of the simulation clock in Logisim, for this part we want to assume that the clock speed is fixed. What we want instead is to create circuitry that takes in the clock and the two-bit values from SW[1] and SW[0] as inputs. The output would be a new clock signal that is used for the hex display to make its digits flash at a slower speed.

Two modules are required to make this happen:

1. A special kind of counter called a **RateDivider** which takes in the 16Hz clock signal and outputs a new clock signal, slowed down to a rate specified by SW[1] and SW[0].

2. A second counter called **DisplayCounter**. This is a hexadecimal counter that is responsible for counting through the values 0 - *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.

The output of the RateDivider generates a slower alternating signal, which is then used as the enable signal of the DisplayCounter module. Every time RateDivider has counted the appropriate number of clock edges, it generates a high output pulse for one clock cycle. For example, Figure 3 shows a timing diagram that produces a 1 Hz pulse signal from a 50 MHz clock (the kind used on the DE1-SOC board). The resulting 1 Hz pulse signal would provide the Enable value for the DisplayCounter module.

In your prelab report, calculate how large a counter would be required to count 50 million clock cycles, as illustrated by Figure 3. How many binary bits would that counter need to represent such a value? **(PRELAB)**
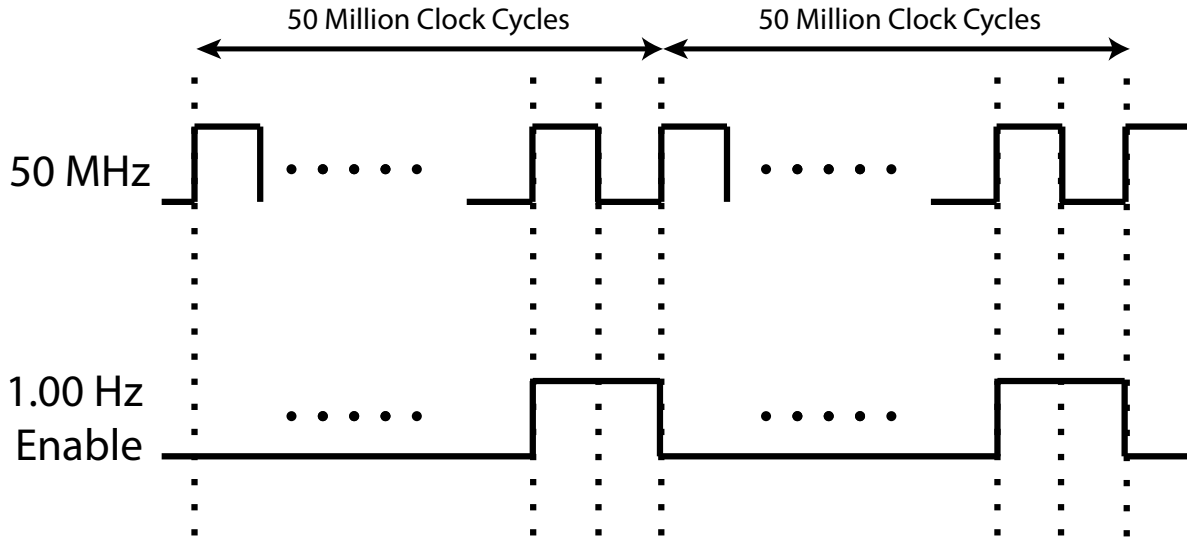
Figure 3: Timing diagram for a 1 Hz enable signal

A common way to count a specified number of clock pulses is to parallel load the RateDivider counter with the appropriate starting value and then count down to zero. For example, if you want to count 50 million clock cycles (as in the above example), simply load the counter with 50 million and subtract one at each time the clock pulses until the counter reaches 0 (strictly speaking, you would load the value 49,999,999). The counter would then be reloaded to the starting value again.

A few more notes about your final circuit:

1. The AND gate on the right side of Figure 2 performs two tasks: it turns on the output *Enable* signal once the counter reaches a certain value and it performs a parallel load of a new counter value (specified by you) on the next clock cycle. This is one way to make the *RateDivider* counts a certain number of clock cycles before turning on the enable signal. Using this idea, changing the frequency of the new clock signal would just involve changing the value you load into the counter.

    - This counter design in Figure 2 is provided for you and you can use it as a model to build your counters.

    - You don't have to follow this model exactly. As long as you use the counter provided by Logisim, you can use whatever other circuitry you prefer to implement the RateDivider behaviour. Be creative! We only provide this idea to get you thinking about what your circuit needs to do.

2. *DisplayCounter* counts 4-bit binary values, incrementing each time its *Enable* input is 1. When the counter reaches 1111, the next increment should reset it to 0000. The output of this *DisplayCounter* should be directed to a seven-segment display so that the TAs can observe the counter result.

    - You can use whichever counter you'd prefer for this part, either the one you created in Part I or Logisim's built-in counter.

For this part of the lab, you need perform the following steps:

1. Draw a schematic of the circuit you wish to build. Mentally step through the usage of this circuit to ensure that your circuit handles the necessary use cases to the best of your understanding. **(PRELAB)**

2. Build the circuit that realizes the behaviour described in your schematic. Your main circuit should have one clock input, one reset input and two external switch inputs (SW[1] and SW[0]) for the RateDivider. The circuit should have a seven-segment display as output. **(PRELAB)**

4

HINT: You should name the inputs and outputs of each module in a way that makes it easy to interpret your simulations (*e.g.* use input/output names from your schematic).

3. Simulate your modules with *Poke*(👆). Choose test cases that make you feel confident about your counter's correctness in preparation for you in-lab demo. Make sure to include a few selected screenshots of these cases when you hand in your prelab. You will also need to think about the best way to simulate this kind of circuit. For example, how many 16 Hz 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. Label your high-level Logisim design with the appropriate input and output names that would be used on the DE1-SoC board (e.g. use $HEX0$ as the seven-segment display from the DisplayCounter while using switches SW_{1-0} to control the rate that these hex digits are updated). You will also need to use additional switches (e.g., the clear signal). **(PRELAB)**

6 Part III

In this part of the lab you are going to *design and 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 eight letters of the alphabet (shown in the table above) and display the Morse code signals for it on an LED (you can find LEDs in *Input/Output > LED*).

To make this happen, you need to create the following:

1. A lookup table (LUT) to store the Morse code sequences for each letter,

2. A shift register to display the Morse signals,

3. A rate divider to set the timing for the shift register.

Let us first look into how we will store the Morse code representation for each letter. A "dot" will cause the LED to flash for 0.5 seconds while a "dash" will cause the LED to flash for 1.5 seconds. Since both of these times are multiples of 0.5 seconds, we will use this to our advantage when representing a Morse code sequence as a sequence of binary numbers.

Each Morse code sequence will be a sequence of 1s and 0s where the 1 indicates when the LED should turn on and the 0 indicates when the LED will turn off. Let's assume that each bit corresponds to a display duration of 0.5 seconds, so a single 1 bit in the sequence 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). The sequence will then use 0 values to separate those signals (*i.e.*, the LED will turn off for 0.5 seconds in between dots and dashes). As a result, the LED could be off either

during a pause (e.g., a transition between a Morse dash and a dot), the beginning/end of a transmission, or if there is no transmission.

Using this representation, the Morse code for letter X could be stored as `1110101011100000`, assuming we use 16-bits to represent it (this is not a required length). To implement the LUT, you will need to use the same approach to fill in the remaining entries in Table 1 with the Morse code binary representation for the other letters specified above (S to Z). **(PRELAB)**

- Note that each letter will require a different minimal number of bits in this representation. However, you will need to make sure all the sequences are the same length because they all need to be loaded into a single shift register to display them on the LED. The shift register will be a fixed length, therefore the length of your binary sequences below will also need to be a single fixed length.

- To figure out what this length will be, you need to figure out the maximum number of bits needed when representing the Morse code sequences for each of the letters (S to Z). Once you have that, adjust all of the binary sequences to match that maximum length. For letters that do not require that maximum number of bits, extend any of the 0 sections of that sequence. Make sure that the last bit of any Morse code representation is set to 0.

Fill in Table 1 below as part of your prelab. You will need to fill in the header of the third column with the sequence length you decided on. You will also need to pad the sequence for letter X with added zeroes as needed to match the sequence length you chose. **(PRELAB)**

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

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

The LUT will store the binary representation of each Morse code pattern and use a multiplexer to determine which pattern is sent to the output bits. The LUT will take in a 3-bit input to specify the letter sequence to display, where 000 denotes the sequence for letter S and 111 denotes the sequence for letter Z. So if each Morse sequence was 16 bits long, the LUT would have a 3-bit input and a 16-bit output. This LUT is asynchronous, so no clock or reset signals are needed.

The next module to consider is the shift register. With the LUT supplying a sequence to display on the LED, we need a module that can display each bit, one at a time, in order. To do that, use a shift register and follow these steps:

1. Load the sequence from the LUT into the shift register (using the shift register's parallel load). Make sure the shift register length is the same as the sequence length from Table 1.

2. Shift the sequence by one bit each time the clock goes high.

3. Have the bit that is shifted out of the register displayed on the LED.

The third and final module that you will need is the rate divider, which will supply the shift register with the appropriate clock signal (one that goes high every 0.5 seconds).

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.

To summarize, once the user presses a button that starts the LED display process you need to load a shift register with the appropriate pattern (for the letter specified by a 3-bit input) and display the Morse
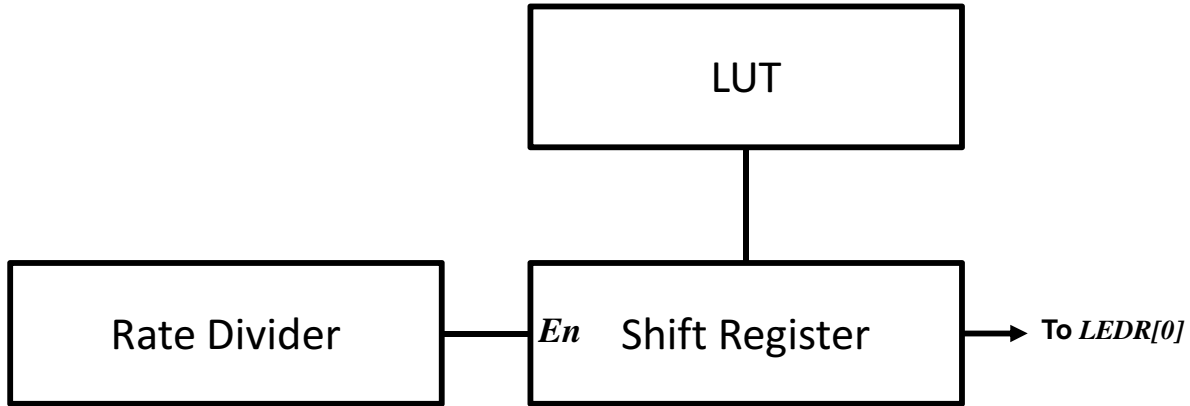
Figure 4: Block diagram of the Morse code circuit

code from that pattern on the LED. Note that you should not display a given letter in a loop (*i.e.*, you should only re-display the same letter if the user presses the start button again).

In the event that you have not completed part II of the lab (*i.e.* you have not been able to implement the 0.5 second enable signal), manually clock your shift register using a switch input signal. This will help you to get partial marks.

Complete the lab by performing 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. Build the circuit in Logisim that realizes the behaviour described in your schematic. **(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).

4. Simulate your LUT with test vectors and your shifter module with *Poke*(👆). Choose test cases that make you feel confident about your LUT and shifter's correctness in preparation for your in-lab demo. You are strongly encouraged to complete this step **before** performing your lab demo. Document the test cases that you considered (especially the corner cases) in your prelab report and include screenshots of the most interesting cases. **(PRELAB)**

5. Label the inputs and outputs of your Logisim design to correspond to the inputs and outputs on the DE1-SoC board. The following table summarizes the inputs and outputs that you will use: **(PRELAB)**

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