LAB 5

COUNTERS AND CLOCKS

In Lab 5, you design counters using T flip-flops. Then, you use these counters to design other circuits that operate at a lower frequency than a global clock. By the end of the lab, you design a circuit that flashes an LED based on Morse Code.

This document describes what you need to prepare and demonstrate for Lab 5. Section 5.3 describes the tasks you must complete *before* your lab session. Section 5.4 describes the tasks you complete *during* your lab session. The next section describes lab logistics in more detail.

5.1 Logistics

Even though you work in pairs during your lab session, you are assessed individually on your Lab Preparation ("pre-lab") and Lab Demonstration ("demo"). All pre-lab exercises are submitted electronically before your lab (see the course website for exact due dates, times, and the submission process). So, **before** each lab, you must read through this document and complete all the pre-lab exercises. During the lab, use your pre-lab designs to help you complete all the required in-lab actions. The more care you put into your pre-lab designs, the faster you will complete your lab.

The Lab Preparation must be completed individually and submitted online by the due date. Follow the steps in Section 5.3 for the pre-lab. Remember to **download the starter files**.

You must upload *every required file* for your pre-lab submission to be complete. But you do *not* need to include images that are not on the list of required files (even if those images are in your lab report). If you have questions about the submission process, please ask ahead of time. The required files for Lab 5's pre-lab (Section 5.3) are:

- Your lab report: `lab5_report.tex`, `lab5_report.pdf` (as generated from the tex file)
- Your digital designs: `lab5.circ`

The Lab Demonstration must be completed during the lab session that you are enrolled in. During a lab demonstration, your TA may ask you to: go through parts of your pre-lab, run and simulate your designs in Logisim, and answer questions related to the lab. You may not receive outside help (e.g., from your partner) when asked a question.

5.2 Marking Scheme

Each lab is worth 4% of your final grade, where you will be graded out of 4 marks for this lab, as follows.

- Prelab: 1 mark
- Part I (in-lab): 0.5 mark
- Part II (in-lab): 1 mark
- Part III (in-lab): 1.5 mark

5.3 Lab Preparation

The pre-lab for Lab 5 consists of three parts. In Part I, you design an 8-bit counter using T flip-flops and hierarchical design. In Part II, you use a counter to produce an enable signal that behaves like a clock, but slower. In Part III, you design a look-up table (LUT) for a subset of Morse Code.

5.3.1 Part I

Consider the circuit in Figure 5.1. 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.

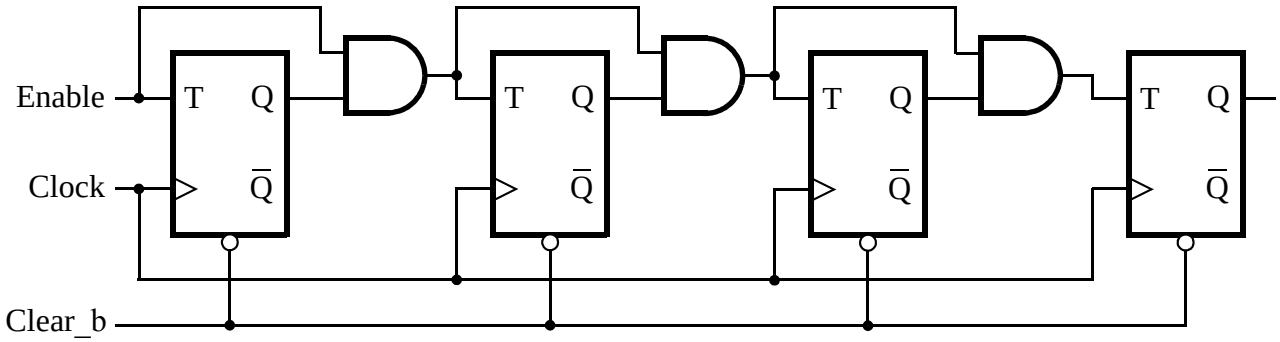

Figure 5.1: A 4-bit counter.

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 and include the required screenshots as asked in the report:

1. Draw the schematic for an 8-bit counter that uses the 4-bit counter as shown in Figure 5.1.
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 (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 5.1.

3. In the starter file called `lab5.circ`, add a new subcircuit called `counter8`. Build the circuit corresponding to your schematic.

   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.

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.

5.3.2 Part II

Another way to specify a counter would be to instantiate a register and find a way to add 1 to the register value every time the clock goes high. Adding 1 can be accomplished using the *Adder* under *Arithmetic*. The constant value 1 can be found in *Wiring* > *Constant*, where the value and the number of data bits for this constant can be set in *Properties* after you click on it.

The Counter Device

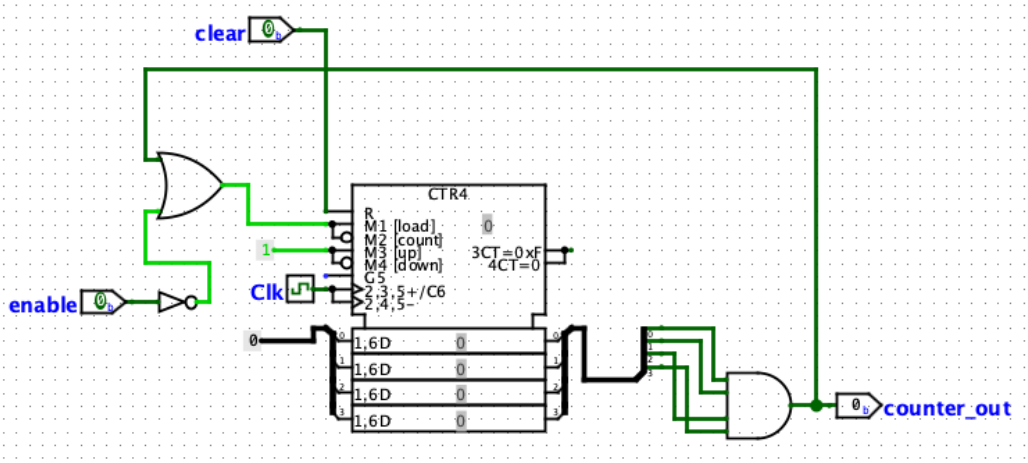

Figure 5.2: Counter in Logisim

Figure 5.2 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. An input of 1 makes the counter load a new multi-bit value (provided in the bottom left of the device) and 0 tells the counter to start counting up or down.

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

4. The inputs called *2,3,5+/C6* and *2,3,5+/C6* should be connected to a Clock signal.

5. The four-bit inputs on the bottom left of the counter take in the value to load into the counter (when the load bit M1 is high) and the value that the counter stores can be seen on the output wires 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.

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

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.

The Rate Divider and the Display Counter

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 7seg 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 (32 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 and/or counter in your circuit should be clocked by a **32Hz** clock signal, provided by Logisim. You will then use a counter to generate the slower clock signals outlined by the table above.

This is different from setting the speed of the simulation clock in Logisim. For this lab we want to assume that your circuit is taking in a clock signal that is fixed at a fast speed and your circuit needs to dynamically slow it down based on the SW[1] and SW[0] inputs.

The output of the circuit you create will be a new clock signal that is used for the 7seg display to make its digits flash at the speed indicated by SW[1] and SW[0] in the table above, assuming that you're starting with the **32Hz** clock provided by Logisim during your simulation.

Two modules are required to make this happen:

1. A special kind of counter called a **RateDivider** which takes in the 32Hz 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 5.3 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 5.3. How many binary bits would that counter need to represent such a value?

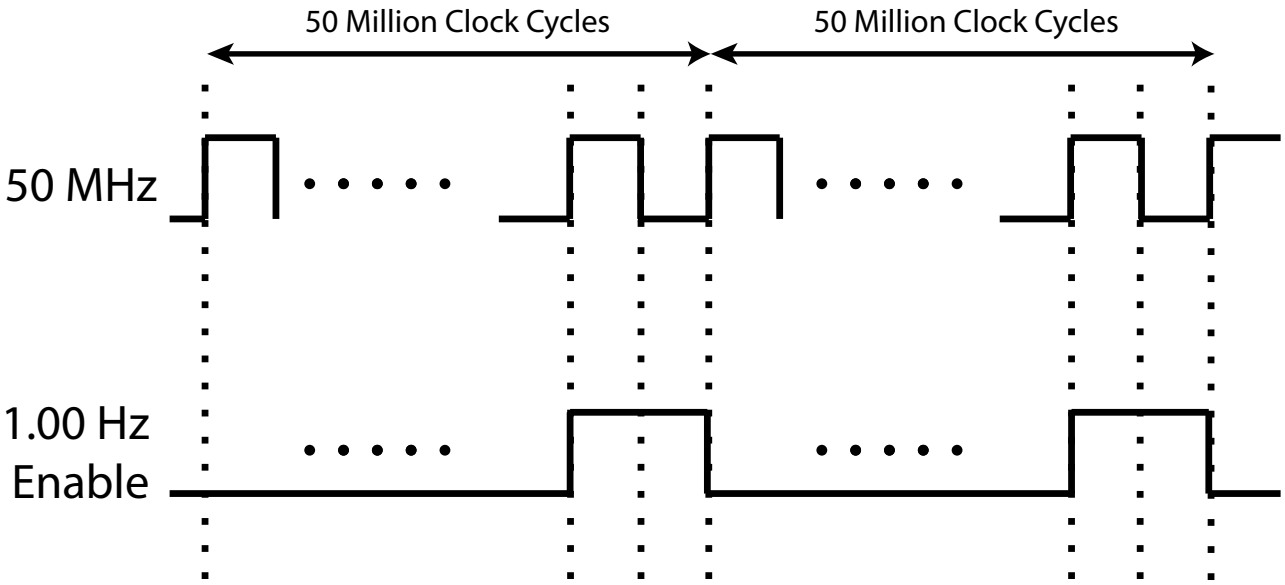

Figure 5.3: Timing diagram for a 1Hz enable signal

A common way to count a specified number of clock pulses is to load a starting value into the RateDivider counter then have the counter count down to zero. For example, if you want to count one second given a 50MHz clock (as in the DE1-SoC example), you can load the RateDivider counter with a value of 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). Once a 50MHz counter has counted down from 50 million, one second has passed and the DisplayCounter can be incremented by one. Then the RateDivider counter would then reload the starting value again. Include the RateDivider's schematics and timing diagram in your prelab report.

A few more notes about your final circuit:

1. The AND gate on the right side of Figure 5.2 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 (but not the only one). 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 5.2 is provided for you and you can use it as a model to build your RateDivider counter (your DisplayCounter will be more straightforward).

    - 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.

Include the schematic of the circuit you built in your prelab report. In particular, show how the outputs of the RateDivider feed into the inputs of the DisplayCounter, and how the high-level inputs and outputs connect to your entire circuit. 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. 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).

Make sure you `Reset Simulation` before starting. Since you are using a "real" clock, Logisim Evolution can automatically run the clock for you. To do this, use `Simulate -> Auto-Tick Enabled` (memorize the keyboard shortcut so you can start and stop the simulation easily). If this is too fast for you, you can manually poke the clock yourself.

Observe the behavior of your 7-segment display for different `Rates`. Does it match what you expect? Use a timer to make sure that the output has the correct frequency (for example, if it should be 1Hz, it's changing once every second).

5.3.3 Part III

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	— — • •

Let us first look into how we store the Morse code representation for each letter. Suppose all the times for dot, dash, and pause are multiples of 0.5 seconds. **Configure (change) the rate divider** so that the shift register shifts every 0.5 seconds for a 32Hz clock. We can use this to our advantage and represent each Morse code as a sequence of bits. Each bit corresponds to a display duration of (e.g., an LED turning on for) 0.5 seconds. Therefore a single 1 bit will correspond to a dot (the LED stays on for 0.5 seconds), while three 1s in a row corresponds 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 stays 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. Observe that a different number of bits is needed for each letter. For letters that do not require the maximum number of bits, you may simply set the last few bits to 0.

Perform the following steps:

1. Write the Morse code binary representation of all letters specified above (S to Z) in a table following the same approach used for X. You must also decide on the maximum number of bits needed when accounting for all letters (S to Z). The last bit of any Morse code representation should be 0.

   Fill in a table with your binary representation of each letter from S to Z.

2. Create a new subcircuit in `lab5.circ` called `MORSE_LUT`. Implement a Look-up Table (LUT) that has a 3-bit input, `Char` to select a letter (i.e., one of S to Z; let 000 correspond to S, 001 correspond to T, and so forth up to 111 for Z). The output, `MorseCode`, should be a multi-bit output whose width is what you decided on in the previous step. Based on the input `Char`, `MorseCode` outputs the binary representation for the specified letter. You can use `Constant`s in Logisim to store these output values.

   Export the subcircuit schematic as an image and include it in your report.

5.4 Lab Demonstration

5.4.1 Part I

Discuss with your partner how you used hierarchical design to connect the 4-bit counters. **Make sure you understand the timing diagrams and, if asked, can generate a timing diagram from scratch.** When you are ready, demonstrate your pre-lab designs to your TA.

5.4.2 Part II

Observe the behavior of your 7-segment display for different `Rate`s. Does it match what you expect? Use a timer to make sure that the output has the correct frequency (for example, if it should be 1Hz, it's changing once every second) When you are done, demonstrate your working circuit to your TA.

5.4.3 Part III

In `lab5.circ`, create a new subcircuit called `morse_code`. It should include: two 1-bit inputs `LoadEnable` and `Enable`, a 3-bit input `CharSelect`, and a Clock from Logisim Evolution. Add a `MORSE_LUT` and `rate_divider` module, as well as a shift register from Logisim Evolution (you can find it under `Memory`). You find Figure 5.4 helpful.

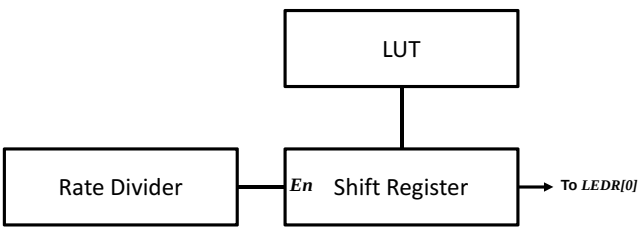

Figure 5.4: A high-level block diagram of the Morse code circuit.

Configure the shift register to match the bit width of your `MORSE_LUT`'s output. Also, remember that you had to **configure (change) the rate divider** so that the shift register shifts every 0.5 seconds for a 32Hz clock. Note that this speed is not covered in your circuit from part 2. Is it?

The output of the circuit should be a single LED that flashes the Morse code being selected by `CharSelect`. Make sure you use a timer to confirm the length of each flash is 0.5 seconds.

Synthesize your final design and download it onto the DE1-SoC board.Make sure you set the frequency in the `Download and Synthesize > Clock Setting` to 32Hz. The `Divider Value` will be set on its own. This divider value will divide the board's clock which is 50MHz by a value to output the 32Hz Clock for your design.

When you are done, demonstrate your working circuit to your TA. If you are unable to get your circuit working on the DE1-SoC board, demo the working design in Logisim for 1 marks instead of 1.5 mark.