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.

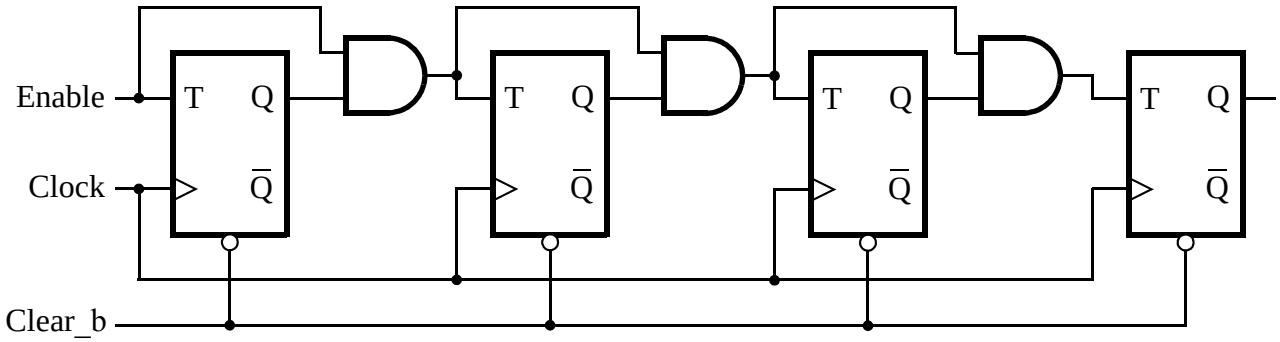

Figure 5.1: A 4-bit counter.

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:

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 an counter would be to instantiate a register and finding 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

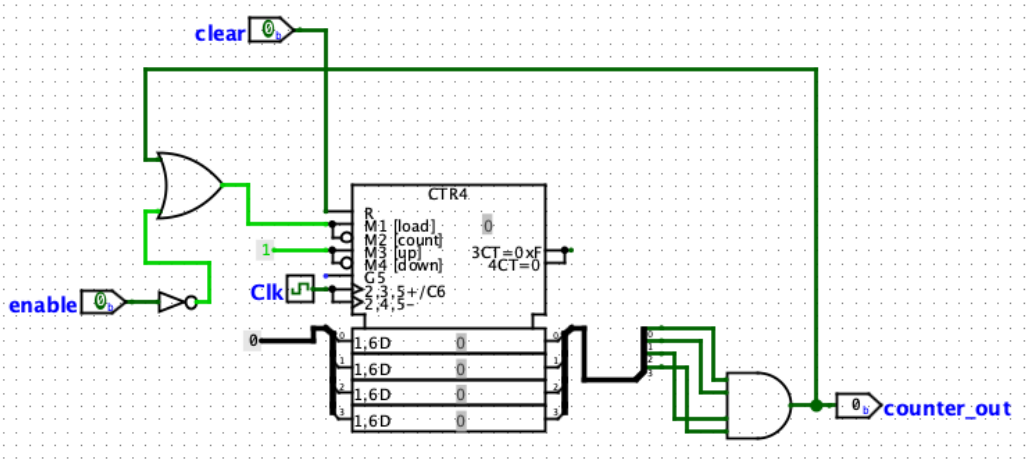

Figure 5.2: Counter in Logisim

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

Typically, a digital system has one global clock. However, sometimes it is desirable to have some circuits operate at a slower speed than the global clock. It is useful to design a subcircuit that helps you run other circuits at a rate that is slower than the clock. Remember, however, that a fully synchronous circuit is one where every flip-flop in your circuit is connected to the *same* clock. So the output of this subcircuit should **not** connect to the clock input of another circuit.

In this part, you design a "rate divider" (sometimes called a frequency divider) that produces a new clock signal. **In this lab, we assume that the global clock is 32Hz** (remember to set this in Logisim Evolution by navigating to: `Simulate` then `Auto-Tick Frequency`). This new clock signal is slower than the global clock, and can be used to drive the *enable* inputs of other circuits that need to operate at a slower rate. One of the rate divider's inputs, `Rate`, dictates how much slower the new clock signal will be (see Table 5.1).

$Rate_1$	$Rate_0$	Frequency (Hz)
0	0	$\frac{f_{global}}{2}$
0	1	$\frac{f_{global}}{32}$
1	0	$\frac{f_{global}}{64}$
1	1	$\frac{f_{global}}{128}$

Table 5.1: The supported frequencies.

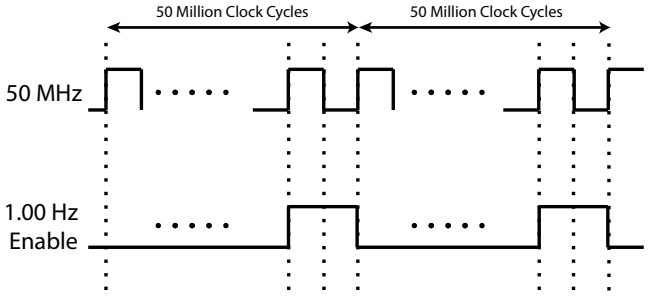

Figure 5.3: Timing of a 1 Hertz enable signal.

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 in the lab rooms). The resulting 1 Hz pulse signal would provide the Enable value for, for example, a counter. 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). Then subtract one from it until it reaches 0. After that reload 49,999,999.

In this lab you count up, not down. First, review the subcircuit `handout_example` in the starter file (same circuit as Figure 5.2). Then, perform the following steps:

1. In the file `lab5.circ`, add a new subcircuit called `rate_divider`. Design a circuit that has 1-bit inputs, `Enable, Reset, Clock`, a 2-bit input, `Rate`, and a 1-bit output `DivOut`. Using Logisim counter, implement a Rate Divider as described above.

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

2. Simulate the your rate divider using `Simulate -> Timing Diagram`. Make sure you `Reset Simulation` before starting. Demonstrate that the circuit's `DivOut` produces a "clock", but at a slower rate than the system's clock (e.g., Figure 5.3).

   Export the timing diagram as an image and include it in your report.

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. 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

The output of the circuit you created `rate_divider` will be a new clock signal that is used for the hex display to make its digits flash at the speed indicated by Rate in the Table 5.1, assuming that you're starting with the **32Hz** (global) clock provided by Logisim during your simulation.

Another module is required to make this happen: 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.

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 before or Logisim's built-in counter.

In `lab5.circ`, merge in a previous lab that includes your `HEX_DECODER` subcircuit (and its associated subcircuits). Then, add a new subcircuit called `demo_rate_divider` that behaves as instructed. Your main and final circuit should have one clock input, one reset input and two external switch inputs (Rate) for the RateDivider. The circuit should have a seven-segment display as output.

CAUTION

You must design a fully synchronous circuit. A fully synchronous circuit is one where every flip-flop in your circuit is connected to the *same* clock. Do **not** connect the output of your `rate_divider` to the clock input of another circuit.

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 behaviour of your 7-segment display for different `Rate`s. Does it match what you expect? When you are ready, demonstrate the simulation 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.

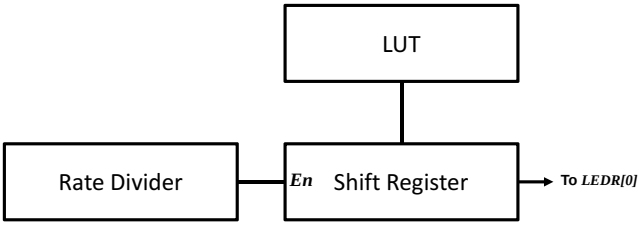

Figure 5.4: A high-level block diagram of the Morse code circuit.

Configure the shift register to match the bit width of your `MORSE_LUT`'s output. Also configure the rate divider to so that the shift register shifts every 0.5 seconds for a 32Hz clock. The output of the circuit should be a single LED that flashes the morse code being selected by `CharSelect`. When you are ready, demonstrate the simulation to your TA.