CSCB58 - Lab 4

Clocks and Counters

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 actual clock rate is much faster. We will also be looking at some features of Quartus that allow you to analyze things you've built that may be helpful when debugging more complicated systems.

Marking Scheme

Prelab	/3
Part I (in-lab)	/1
Part II (in-lab)	/1
Part III (in-lab)	/2
Clean work-space with all materials returned to their original state	/1
TOTAL	/8

Write your name, UTorID, and student ID:

Name: _____

Student ID: _____

UTorID: _____

Write your partner's name, UTorID, and student ID:

Partner name: _____

Partner student ID: _____

Partner UTorID: _____

> **You may run into bounce problems using KEYs. If so, you are welcome to try using any of the keys if you find they preform better, or simply change to one of the switches.**

Part I

Consider the circuit in Figure 1. It is a 4-bit synchronous counter that uses four T-type flip-flops to implement a counter. The counter increments its value on each positive edge of the clock if the `Enable` signal is asserted. The counter is reset to 0 by setting the `Clear_b` signal low – it is an **active-low asynchronous** clear. You are to implement an 8-bit counter of this type.
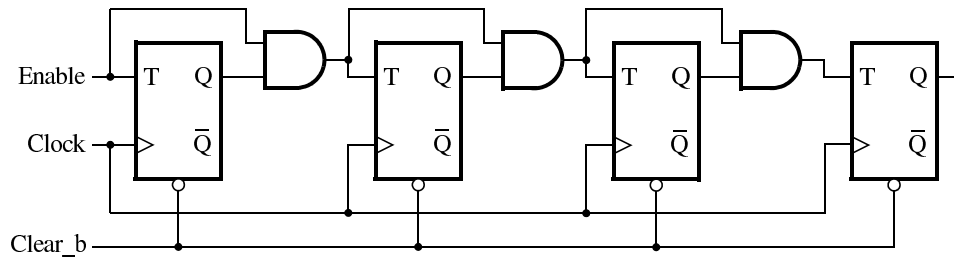

Figure 1: A 4-bit counter.

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 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.

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, 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.

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 $Q_7Q_6Q_5Q_4Q_3Q_2Q_1Q_0$ they correspond to. **(PRELAB)**

3. Write the Verilog corresponding to your schematic. Your code should use a module that is instantiated eight times to create the counter. **(PRELAB)**

4. Compile the circuit in Quartus and answer the following questions:

   (a) How many logic elements are used to implement your circuit?
   In Quartus, look at your `Logic Utilization (in ALMs – Adaptive Logic Modules)` in the Fitter Report found by double clicking `Compile Design -> Fitter -> View Report` in the Tasks pane on the left. This 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?

   (b) What is the maximum frequency, F_{max}, at which your circuit can be operated? To find this maximum frequency, compile your code in Quartus, and then run the TimeQuest Timing Analyser. The F_{max} summary can be found in on the left in `Reports -> Datasheets -> Report Fmax Summary`. Refer to the *Using TimeQuest Timing Analyzer* document found on the Intel/Altera website for more information:
   https://fpgauniversity.intel.com/redirect/materials?id=/pub/Intel_Material/
   16.1/Tutorials/Verilog/Timequest.pdf

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 count as your circuit operates. Simulate your circuit to ensure that you have done this correctly.

6. Use the Quartus II RTL Viewer to see how the Quartus II 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?

7. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit. Demonstrate the working circuit to your TA.

Lab continues in the next page...

Part II

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

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

Figure 2 shows an example of a code fragment of a counter that counts from 0 to F in hexadecimal. The counter also has a synchronous clear (`Clear_b`), a parallel load feature (`ParLoad`), and an enable input (`Enable`) to turn the counting on and off.

```
reg [3:0] q;                 // declare q
wire [3:0] d;                // declare d

always @(posedge clock)      // triggered every time clock rises
begin
  if (Clear_b == 1'b0)       // when Clear_b is 0...
    q <= 0;                  // set q to 0
  else if (ParLoad == 1'b1)  // ...otherwise, check if parallel load
    q <= d;                  // load d
  else if (q == 4'b1111)     // ...otherwise if q is the maximum counter value
    q <= 0;                  // reset q to 0
  else if (Enable == 1'b1)   // ...otherwise update q (only when Enable is 1)
    q <= q + 1'b1;           // increment q
//  q <= q - 1'b1;           // alternatively, decrement q
end
```

Figure 2: Example counter code fragment

Observe that `q` is declared as a 4-bit value making this a 4-bit counter. The check for the maximum value is not necessary in the example above. Why? If you wanted this 4-bit counter to count from 0-9, what would you change?

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 DE2 board. At this speed, what do you expect to see on the display? (HINT: compute the period of that clock. If it has 50,000,000 cycles per second, what is the length of each cycle?)

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 counter, let us call it **RateDivider**, that is also clocked with the 50 MHz clock. The output of RateDivider can be used as part of a circuit to create pulses at the required rates. Every time RateDivider has counted the appropriate number of clock edges, a pulse should 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 should 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? How many bits would you need to represent such a value?
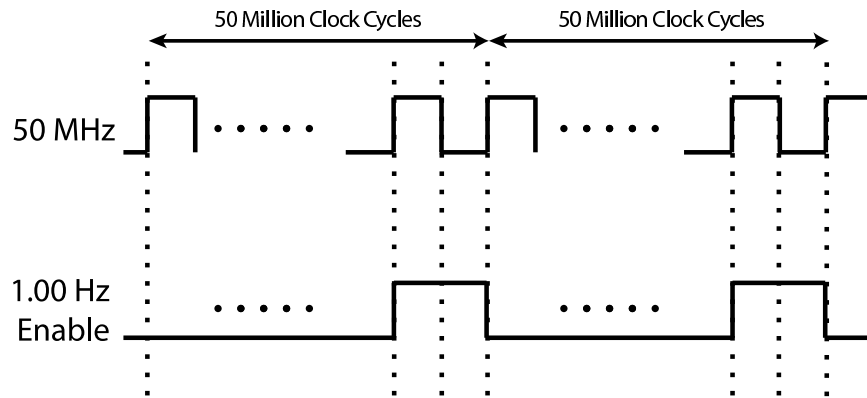
Figure 3: Timing diagram for a 1 Hz enable signal

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. (Why subtract 1?) 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: the above example assumes that RateDivider is a four-bit counter. You will need to adjust its width depending on the counter width you use.

Recall that an `Enable` signal determines whether a flip flop, register, or counter will change on a clock pulse. The pulses from the rate divider can be used to drive an `Enable` signal on the hexadecimal counter (let us call this the **DisplayCounter**) that is counting from 0 through F.

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.

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 the clock and the two switches as inputs. **(PRELAB)**

   In addition to switches SW_{1-0} used to control the rate the 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 DE2 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 a port called `CLOCK_50` in your top-level module.

3. Compile the project.

4. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit. And demonstrate your working code to the TA.

Part III

In this part of the exercise you are 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, starting from A, the first eight letters of the alphabet have the following representation:

A	• —
B	— • • •
C	— • — •
D	— • •
E	•
F	• • — •
G	— — •
H	• • • •

Your circuit should take as input one of the eight letters of the alphabet starting from A (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 A, 001 for B, etc.). Use 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 need a shift register, a rate divider similar to what you used in Part II, and a lookup table (LUT) to store the Morse codes. Let us first look into how we will store the Morse code representation for each letter. Since the minimum time for a pulse (dot) or a space is 0.5 seconds, we will set each 0 or 1 (i.e., a single bit) to 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) correspond to a dash (the LED should stay on for $3 \times 0.5 = 1.5$ 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 F would be stored as: 1010111010000000, assuming we use 16-bits to represent it: dot, pause, dot, pause, dash, pause, dot, pause. Write the Morse code binary representation of all eight letters (A to H) following the same approach. You will observe that a different number of bits is needed for each letter. You should figure out the minimum number of bits needed accounting for all letters, since all letters need to be stored using the same pattern length. The last bit of any Morse code representation should be 0.

Lab continues in next page...

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 F with as many zeros as needed based on the pattern length you chose. **(PRELAB)**

Letter	Morse Code	Pattern Representation (pattern length is _____ bits)
A	• —	
B	— • • •	
C	— • — •	
D	— • •	
E	•	
F	• • — •	1010111010_____
G	— — •	
H	• • • •	

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 one we want, we need to display the pattern on the LED one bit at a time for a duration of 0.5 seconds per bit. To do that you need to load a shift register in parallel with that pattern. The register should have the appropriate bit-width. 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). 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 all *bit-width* bits of 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).

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), for part marks you may manually clock your shift register using one of the KEY inputs.

Perform the following steps.

1. Use Table 1 to determine your codes and bit-width.

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. **(PRELAB)**

3. Write a Verilog module that realizes the behaviour described in your schematic. **(PRELAB)**

4. Download the compiled circuit into the FPGA. Test the functionality of the circuit. Demonstrate your working code to the TA.

5. Submit your .v files for all parts of this lab to Quercus.