Laboratory Exercise 5

Clocks and Counters
October 15, 2018

The purpose of this exercise is to learn how to create counters and to be able to control the sequencing of operations when the actual clock rate is much faster than the rate the operations are occurring.

Preparation Before the Lab

You are required to complete Parts I to III of the lab by writing and testing Verilog code and compiling it with Quartus. Show your schematics, Verilog, and simulations for Parts I to III to the teaching assistants. You must simulate your circuit with ModelSim (using reasonable test vectors you can justify).

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.

Part I

Consider the circuit in Figure 1. It is a 4-bit synchronous counter (text Section 5.9.2) that uses four T-type flip-flops (text Section 5.5). 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.

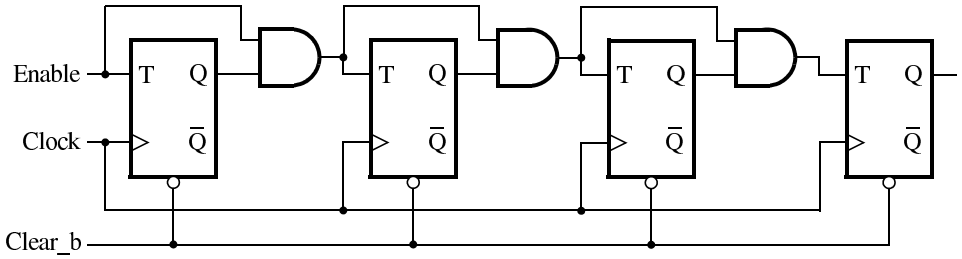

Figure 1: A 4-bit counter.

Perform the following steps:

1. Draw the schematic for an 8-bit counter using the same structure as shown in Figure 1.
2. Write a Verilog module for a T-type flip flop (text Section 5.5).
3. Write the Verilog module corresponding to your schematic. Your code should use your T-type flip-flop module that is instantiated eight times.
4. Simulate your circuit to verify its correctness.
5. Compile the circuit. To answer the following questions open the compilation report and look under *Fitter* for resource utilization and *TimeQuest Timing Analyzer* for F_{max} (open the compilation report and look under Timing Analyzer $\rightarrow$ Slow 1100 mV 85C Model $\rightarrow$ F_{max}).

How many logic elements (LEs) are used to implement your circuit? 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?

What is the maximum frequency, F_{max}, at which your circuit can be operated?

6. Use the pushbutton 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.

7. Use the Quartus RTL Viewer to see how the Quartus software synthesized your circuit. What are the differences in comparison with Figure 1?

8. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit.

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:

$$\mathrm{Q} <= \mathrm{Q} + 1;$$

An example code fragment is shown in Figure 2 of a counter that counts from hexadecimal values 0 to F. The counter also has a synchronous clear, a parallel load feature, and an enable input 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;                   // q is set to 0
   else if (ParLoad == 1'b1)    // Check if parallel load
      q <= d;                   // load d
   else if (q == 4'b1111)       // when q is the maximum value for the counter
      q <= 0;                   // q reset to 0
   else if (Enable == 1'b1)     // increment q only when Enable is 1
      q <= q + 1;               // increment 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 do?

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
0	1	4 Hz
1	0	2 Hz
1	1	1 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?

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, 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 pulses, a pulse should be generated for one clock cycle. Figure 3 shows a timing diagram for a 1 Hz Enable signal with respect to a 50 MHz clock. How large a counter is required to count 50 million clock cycles?

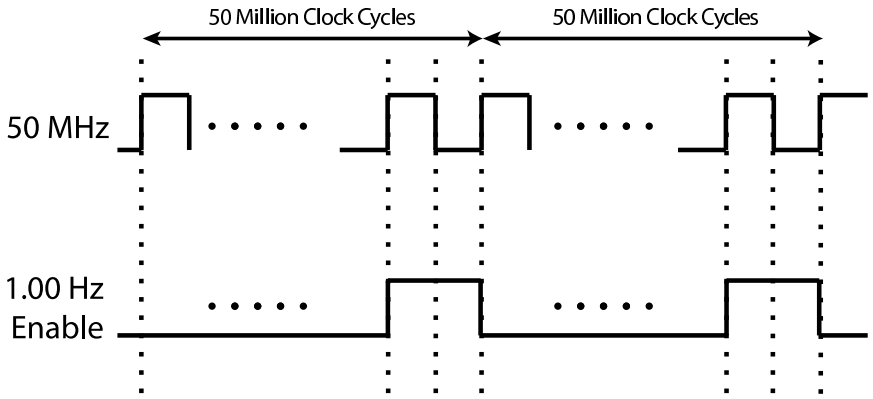

Figure 3: Timing diagram for a 1 Hz enable signal

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 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 an *Enable* signal on the hexadecimal counter, 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 a clock pulse.

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.
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.

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 a 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 show this to the TA as part of your preparation. 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?

4. Compile the project.

5. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit.

Part III

In this part of the exercise you are to implement a Morse code encoder using a lookup table (LUT) to store the codes, a shift register (text Section 5.8.1), and a rate divider similar to what you used in Part II.

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 S, the first eight letters of the 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, i.e., flash, the Morse code for it on $LEDR_0$. Use switches SW_{2-0} and pushbuttons 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. Pushbutton KEY_0 should function as an asynchronous reset.

Hint: You can encode the pattern for each letter using a sequence of 1's and 0's. Since your minimum time is 0.5 seconds, set a 0 or 1 in your code to be 0.5 seconds in duration. This means that a single 0 is a pause or off, a single 1 is a dot, and 111 is a dash. Then read each 0 or 1 individually out of a shift register at 0.5 seconds per read. You should have observed that the codes are different lengths. You may assume that all letters can be stored using a single pattern length, i.e., all patterns stored in the LUT use the same number of bits. For example, the pattern for U would be stored as `1010111000000000` assuming that you are using 16-bit patterns. How many bits do you really need?

The LUT 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. A shift register is first loaded in parallel with a pattern and then the pattern is shifted out of the register, one bit at a time, to be displayed for the appropriate interval.

Perform the following steps.

1. 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.
2. Write a Verilog module that realizes the behaviour described in your schematic.
3. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct.
4. Compile the project.
5. Download the compiled circuit into the FPGA. Test the functionality of the circuit.