Laboratory Exercise 5

Clocks and Counters

Revision of October 20, 2021

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.

1 Work Flow

For each part of the lab you should begin by writing and testing Verilog code and compiling it with Quartus. You should be prepared to show schematics, Verilog, and simulations to your TA, if requested. You must simulate your circuit with ModelSim using reasonable test vectors written in the format used in Lab 2 for the simulation files.

2 Part I

Consider the circuit in Figure 1. It is a 4-bit synchronous counter (textbook Section 5.9.2) that uses four T-type flip-flops (textbook Section 5.5). The value of the counter is comprised of the Q outputs of the T flip-flops. The least significant bit is on the left in Figure 1. 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.

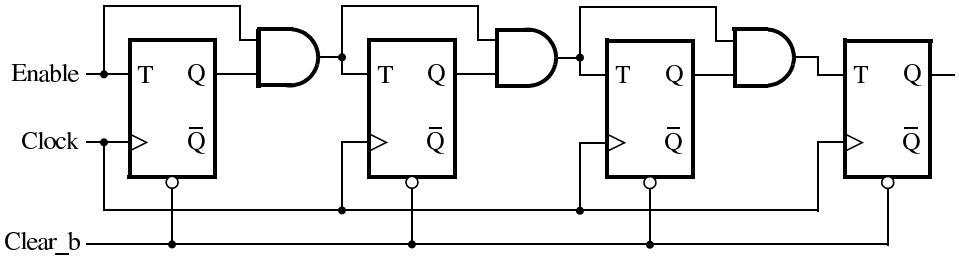

Figure 1: A 4-bit counter.

2.1 What to Do

The top-level module of your design should have the following signature declaration:

```
module part1(Clock, Enable, Clear_b, CounterValue);
```

The `CounterValue` is the value of the counter comprised of the Q outputs of all of the T flip-flops where `CounterValue[7]` is the most significant bit and `CounterValue[0]` is the least significant bit of the counter output.

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 (textbook 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 counter with ModelSim to satisfy yourself that your circuit is working. Be prepared to justify that your test cases are enough to give confidence that your circuit is working. When you are satisfied with your simulations, you can submit to the Automarker.
5. Create a new Quartus project for your circuit. You will need a top-level module to make connections from the instantiation of your `part1` module to the pushbuttons, switches and HEX displays of the DE1-SoC board. 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 new circuit with the DE1-SoC connections with ModelSim to ensure that you have done the connections correctly.
6. Synthesize the circuit with Quartus. To answer the following questions open the compilation report and look under *Fitter* for resource utilization and *TimeQuest Timing Analyzer* for F_{max}.

   How many logic elements (LEs) are used to implement your circuit? For an Intel device, the name of an LE is the ALM (Adaptive Logic Module). 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?
7. Use the Quartus RTL Viewer to see how the Quartus software synthesized your Verilog into a circuit. What are the differences between the circuit synthesized by Quartus and Figure 1?

8. Compile the project to generate a bitstream to make sure your code can at least be synthesized.

9. If you have a board, download the compiled circuit into the FPGA chip. Test the functionality of the circuit.

10. If you do not have a board, you can use *fake_fpga* to test that your circuit behaves as expected.

3 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:

$$Q <= Q + 1;$$

Clearly, this is much easier than what you did in Part I. Part I is done to show you the fundamentals of how a counter circuit works, but when using Verilog, we can use the above construct and let the synthesis tool create the actual circuit. It is also much easier to add many features to your counters showing the benefits of Verilog.

An example code fragment is shown in Figure 2 of a counter that counts in hexadecimal going through the values 0 to F repeatedly. The counter also has a parallel load feature (textbook Section 5.9.3), and a synchronous clear and an enable input (textbook Section 5.9.2) 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?

3.1 The Circuit to Build

Design and implement a circuit using counters that repeatedly outputs the hexadecimal values 0 through F at an output called `CounterValue`. The circuit is driven by an input clock called `ClockIn`. There is an active high reset signal called `Reset` to clear or initialize counters as required. The rate at which the digits change is set by a 2-bit input called `Speed` according to the following table:

Speed[1]	Speed[0]	CountRate	Description
0	0	Full	Once every clock period
0	1	1 Hz	Once a second
1	0	0.5 Hz	Once every two seconds
1	1	0.25 Hz	Once every four seconds

If you use the DE1-SoC board and its 50 MHz clock, Full speed means that the display flashes at 50 MHz, i.e., 50 million times a second. 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 `ClockIn` clock signal.

Now, assume that the circuit will run on the DE1-SoC board and `ClockIn` is driven by the 50 MHz clock available on the board.

To derive the slower flashing rates you should use a counter, call it `RateDivider`, that is also clocked with `ClockIn`. 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 assuming that `ClockIn` is driven by a 50 MHz clock. How large a counter is required to count 50 million clock cycles?

A common way to provide the ability to change the number of pulses counted is to parallel load the counter with the appropriate starting value and count down to zero. With this approach, the end condition is always 0, and you can just load the counter with different starting values depending on the period you want to count. For example, if you want to count 50 million clock cycles, load the counter with 50 million - 1. Why subtract 1?

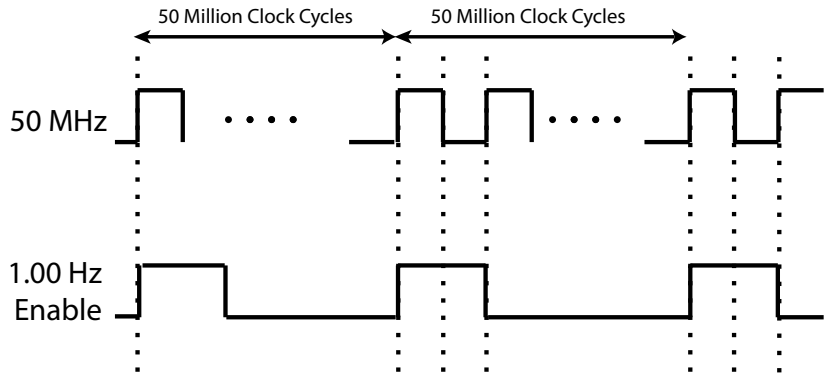

Figure 3: Timing diagram for a 1 Hz enable signal

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 the size of the counter depending on the counter width you use.

Assume the hexadecimal counter is called `DisplayCounter`. The `Enable` pulses generated using `RateDivider` can be used to drive an enable signal on `DisplayCounter` called `EnableDC`. 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 determined by the `Speed` input so that `Enable` pulses are generated at the required frequency. `DisplayCounter` counts through the hexadecimal values, but only increments when its `EnableDC` 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.

3.2 Automarking and *fake_fpga*

If you have an actual board available, you can continue your design assuming a 50 MHz clock because you will have an actual 50 MHz clock on your board. If you connect your `ClockIn` signal to the pin called `CLOCK_50` you will access the 50 MHz clock.

If you are using the *automarker* or *fake_fpga* then you will have to do your design differently.

Automarker Assume that when you connect to `CLOCK_50` that you will get a 500 Hz clock and you will have to adjust your counter values accordingly. This is because the automarker uses simulation and is checking assuming a 500 Hz clock is used.

fake_fpga When you connect to `CLOCK_50` on *fake_fpga* you will get a 5000 Hz (5 KHz) clock. This means that if you want to see a 1 Hz flashing rate on *fake_fpga* you will need to use constants corresponding to a 5000 Hz (5 KHz) clock in your circuit.

3.3 What to Do

The top-level module of your design should have the following declaration:

```
module part2(ClockIn, Reset, Speed, CounterValue);
```

Important: To match the expectations of the Automarker, you must follow these requirements:

1. Your `RateDivider` counter must be implemented so that it counts down to 0 and generates an enable pulse when it reaches 0.
2. For the `RateDivider` counter, if you change the `Speed` in middle of counting down, the `RateDivider` counter should **continue to count down to 0** and load the new frequency after generating the enable signal.
3. When the `RateDivider` counter is reset, it should be reset to the full speed value.
4. Remember that CLOCK_50 gives you a 500 Hz clock when using the automarker.

Perform the following steps.

1. Draw a schematic of the circuit you wish to build. This circuit is more complex so it is especially important that you draw a schematic to help you understand how the circuit should work. Work through the functionality of your circuit on the schematic 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 `ClockIn`, `Reset` and `Speed` as inputs. The only output is `CounterValue`. Your Verilog code should use the same names for the wires and instances as shown in your schematic.

3. Simulate your `part2` module with ModelSim to satisfy yourself that your circuit is working. Be prepared to justify that your test cases are enough to give confidence that your circuit is working.

   NOTE 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? Does it seem reasonable to simulate that many pulses? To help answer this, assume it takes 0.001 seconds to simulate one clock pulse. How long will the simulation take? How can you deal with this situation?

   When you are satisfied with your simulations, you can submit to the Automarker, but make sure you have followed the Important requirements given at the beginning of this section and paid attention to Section 3.2.

4. Create a new Quartus project for your circuit. You will need a top-level module to make connections from the instantiation of your `part2` module to the switches and HEX displays of the DE1-SoC board. Use two switches, SW_1 and SW_0, to drive the `Speed` input of your module. You should use SW[9] to drive the `Reset` input of your module.

   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.

   Simulate your new circuit with the DE1-SoC connections with ModelSim to ensure that you have done the connections correctly.

5. Compile the project to generate a bitstream to make sure your code can at least be synthesized.

6. If you have a board, download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the various inputs and observing the outputs.

7. If you do not have a board, you can use *fake_fpga* to test that your circuit behaves as expected. Remember to account for the different clock speeds.

4 Part III

In this part of the exercise you are to implement a Morse code encoder using a lookup table (LUT[1]) to store the codes, a shift register (textbook Section 5.8.1), and a rate divider similar to what you used in Part II.

[1]Do not confuse this with the LUTs in the FPGA.

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, i.e., flash, the Morse code for it at the output `DotDashOut` using short and long pulses. The letter is selected by the `Letter` input using 000 for A, 001 for B, etc. The Morse code for the letter is output when the `Start` signal is set high using 0.5-second pulses to represent dots, and 1.5-second pulses to represent dashes. The time between pulses is 0.5 seconds. The `Resetn` input is an active-low asynchronous reset. If no signals have been changed (i.e., no new `Letters` have been selected, `Resetn` is high and `Start` is low), the `DotDashOut` should stay the same.

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 A would be stored as `1011100000000000` assuming that you are using 16-bit patterns. How many bits do you really need? (Note: The automarker expects 12 bits to be used for representing the patterns for the letters.)

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.

4.1 What to Do

The top-level module of your design should have the following signature declaration:

```
module part3(ClockIn, Resetn, Start, Letter, DotDashOut);
```

Important To match the expectations of the Automarker, you must follow this requirement:

1. When you set the `Start` signal while the previous Morse code sequence is being output, the Morse code specified by the new `Letter` value should be displayed. This means that the previous code sequence is interrupted and you should begin displaying the new code from the start. The `Start` signal will be high for one cycle in the tester.

As in the previous part, if you are using the Automarker or *fake_fpga*, refer to Section 3.2 regarding the constants (clock frequencies) to use for your timing calculations.

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. Your circuit should have `ClockIn`, `Resetn`, `Start` and `Letter` as inputs. The only output is `DotDashOut`. Your Verilog code should use the same names for the wires and instances as shown in your schematic.

3. Simulate your `part3` module with ModelSim to satisfy yourself that your circuit is working. Be prepared to justify that your test cases are enough to give confidence that your circuit is working.

   When you are satisfied with your simulations, you can submit to the Automarker, but make sure you have followed the Important requirement given at the beginning of this section and paid attention to Section 3.2.

4. Create a new Quartus project for your circuit. You will need a top-level module to make connections from the instantiation of your `part3` module to the switches, pushbuttons and LED displays of the DE1-SoC board. When a user presses KEY_1, the circuit should flash on $LEDR_0$ the Morse code for a letter specified by SW_{2-0}. Pushbutton KEY_0 should be used as the active-low asynchronous reset. Use `CLOCK_50` as the input to `ClockIn`. Remember that the pushbuttons on the DE1-SoC board are active low.

   Simulate your new circuit with the DE1-SoC connections with ModelSim to ensure that you have done the connections correctly.

5. Compile the project to generate a bitstream to make sure your code can at least be synthesized.

6. If you have a board, download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the various inputs and observing the outputs.

7. If you do not have a board, you can use *fake_fpga* to test that your circuit behaves as expected. Remember to acccount for the different clock speeds.

5 Submission

When submitting to the Automarker make sure you have modules declared as shown below as the Automarker will be looking for modules with these exact signatures. Ensure you have properly followed Section 3.2 prior to submitting your code.

5.1 Part I

For Part I, you need to submit a file named `part1.v` with the following module in it:

1. `module part1(Clock, Enable, Clear_b, CounterValue);`

5.2 Part II

For Part II, you need to submit a file named `part2.v` with the following module in it:

1. `module part2(ClockIn, Reset, Speed, CounterValue);`

5.3 Part III

For Part III, you need to submit a file named `part3.v` with the following module in it:

1. `module part3(ClockIn, Resetn, Start, Letter, DotDashOut);`