Laboratory Exercise 5

Clocks and Counters

Revision of October 18, 2023

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. Each part of this lab uses flip flops with an **active-high, synchronous** reset.

WARNING: Parts II and III of this lab are complex designs. You are **strongly encouraged** to read the lab document carefully, more than once to make sure you understand what you must do. Then, design and test your code as you go. Writing a lot of code and testing it all at the end can lead to you wasting a lot of time.

This is also a good lab to try to run part 1 and 3 on the FPGA, even if it is not part of the demo. You should try to complete as many parts as you can before your lab session and run them on the FPGA during your lab session. If you cannot do this, try to run your code on the FPGAs in the drop-in lab (BA3135) which you can access 24/7 using your T-card.

1 Part I

Consider the circuit in Figure 1. It is a 4-bit synchronous counter that uses four T-type flip-flops. The value of the counter comprises 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.

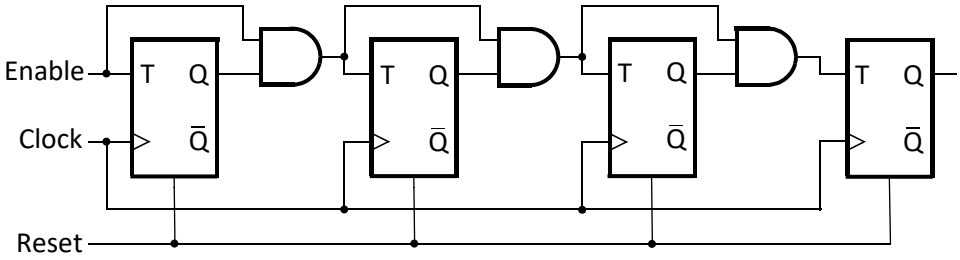

Figure 1: A 4-bit counter using T flip flops.

1.1 What to Do

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

```
module part1 (
    input Clock,
    input Enable,
    input Reset,
    output [7:0] 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. Prior to coming to the lab, write a Verilog module for a T-type flip flop. Recall from lecture, that a T-flip flop is built from a D Flip Flop and an XOR gate. **NOTE:** Do not name your module `tff` as that is a reserved Quartus keyword.
2. Write the Verilog module corresponding to your schematic. Your code should instantiate your T-type flip-flop module eight times.
3. Prior to coming to the lab, simulate your counter with ModelSim to satisfy yourself that your circuit is working.
4. When you are satisfied with your simulations, you can submit to the Automarker.

1.2 Running on the FPGA

If you wish to run your design on an FPGA, you may use the mapping provided below.

module Port Name	Direction	DE1-SoC Pin Name
`Clock`	Input	KEY[0]
`Enable`	Input	SW[0]
`Reset`	Input	SW[1]
`CounterValue`	Output	LEDR[7:0] and HEX1, HEX0

Table 1: Module port mapping to DE1-SoC/DE10-Lite pin names

2 Part II

In this part, you will design a counter that continuously outputs the hexadecimal values 0 through F, to an output called `CounterValue`. The rate at which the outputs change will be configured using the `Speed` input. There is also a CLOCK_FREQUENCY Parameter with a range of values of **100-50000000** specifying the ClockIn Frequency in Hz. **The automarker may test the project at 100Hz, 500Hz, or 2000Hz**, and will supply this parameter to say which frequency it is testing at. You should use 50000000Hz for the demo. Your code should account for this parameter.

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

```
module part2
#(parameter CLOCK_FREQUENCY = 50000000)(
    input ClockIn,
    input Reset,
    input [1:0] Speed,
    output [3:0] CounterValue
);
```

In this section, you will build your counter with D-flip flops. Recall, you can build a counter by using a register and adding 1 to its value (textbook Section 5.13, figure 5.52).

```
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.

Once you've built the counter, you will need to create a module that will increment the counter at different speeds.

The rate at which the numbers change based on the `Speed` input is given in the following table:

Speed[1:0]	CountRate	Description
00	Full	Once every clock period
01	1 Hz	Once a second
10	0.5 Hz	Once every two seconds
11	0.25 Hz	Once every four seconds

For example, let's say `ClockIn` is 50 MHz, which is the clock available on the FPGA boards. In this case, *full speed* means that the display flashes at 50 MHz, i.e., 50 million times a

second. **Question**: If you ran a counter at 50 MHz and displayed the value on a HEX, what do you think you would see?

Running the counter at full speed is easy; you simply increment/decrement the counter by 1 every cycle. Supporting different speeds requires writing two modules: a `Rate Divider` and a `Display Counter`.

Note: Since this design uses two complex modules, it is important to draw a well-labelled schematic before writing any code. **You should prepare this schematic prior to coming to lab.** Your schematic should include block diagrams of the `part2` module and show the `Rate Divider` and `Display Counter` modules inside it. You must have this schematic ready to show during your lab session.

2.1 Rate Divider

Let's start with the `Rate divider` module. A `Rate Divider` is a counter that is used to create a slow *pulse* given a faster clock signal. A *pulse* is when a signal is 1 for a single clock period but is 0 at other times. Figure 2 shows a timing diagram for a 1 Hz pulse for an input clock of 50 MHz.

Creating a pulse. To create a pulse, you need to count N-cycles of your input clock before generating your pulse. For example, to create a 5 MHz pulse given a 50 MHz clock, you count 10 cycles of the 50 MHz clock. As you can see, the sizing of your counter depends on ratio between your `clock frequency` and your `pulse frequency`. **Question**: For the case shown in Figure 2, how many bits should the counter be?

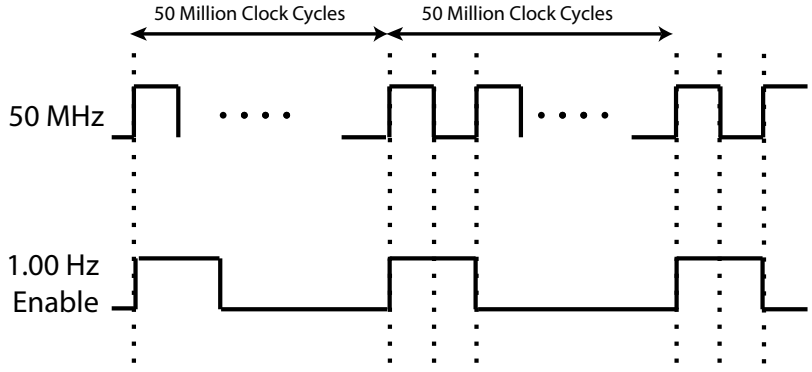

Figure 2: Timing diagram for a 1 Hz enable signal

Outputting the pulse when the counter is zero can be done using a *conditional assign statement* like:

```
assign pulse = (RateDividerCount == 'b0)?'1:'0;
```

2.2 Making the Rate Divider flexible.

To make the `Rate Divider` flexible, we want it to support two things: 1) counting different numbers of clock cycles and 2) different clock frequencies.

Counting different numer of clock cycles. A common way to do this 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. **Question**: Why subtract 1? Do you think it would make a difference in the above case? What about the case of creating a 10 MHz clock from 50 MHz clock?

Supporting different frequencies. To support different clock frequencies, we will use parameters, which you learned about in Lab 3. We pass in a parameter, called `CLOCK_FREQUENCY`, to our `Rate Divider` module to specify the frequency of our clock. The FPGAs use a clock frequency of 50 MHz, the tester uses a frequency of **500Hz**, and **the automarker may test the project at 100Hz, 500Hz, or 2000Hz**, and will supply this parameter to say which frequency it is testing at. You should use, 50000000Hz for the demo. Your code should account for this parameter. You must calculate how many clock periods you have to count to support the different `Speed` values, for a given clock frequency as specified by the CLOCK_FREQUENCY parameter.

Sizing your counter. A flexible Rate Divider means that sizing your counter is non-trivial. One approach is to make your counter big enough to fit the largest number you will count to. For counting smaller values, the upper bits of your counter will simply be unused. Another, more efficient, way is to size the counter based on the `CLOCK_FREQUENCY` parameter. For example, to count N values, you will need $\lceil log_2(N) \rceil$-bits. You can then use the built-in `$clog2()` function in Verilog, to use the exact right counter size for your design. However, it is also valid to use a counter size that is large enough to work with any value within the range of allowed CLOCK_FREQUENCY values.

2.2.1 Building the final Rate Divider.

Now that you know how to build a flexible `Rate Divider`, you can implement it to have the following declaration.

```
module RateDivider
#(parameter CLOCK_FREQUENCY = 50000000) (
    input ClockIn,
    input Reset,
    input [1:0] Speed,
    output Enable
);
```

You can then instantiate this module in the part2 module as shown:

```
RateDivider #(.CLOCK_FREQUENCY = CLOCK_FREQUENCY) RDInst ( .ClockIn=... );
```

Important: To work with the Automarker, your `Rate Divider` must follow these requirements:

1. Your counter must count down to 0 and generates an enable pulse when it reaches 0.
2. If `Speed` changes while counting down, the counter should **continue to count down to 0** and only change speed after generating the enable signal.

2.2.2 Simulating the Rate Divider.

Before proceeding, you are **strongly encouraged** to write a separate `do` file to test your `Rate Divider` module and make sure it works before proceeding. Your TA may ask to see this do file prior to helping you debug your circuit.

When simulating the `Rate Divider` module, pay attention to the number of cycles you have to simulate to check that the `Enable` pulse is working correctly. Also, it would take too long to simulate millions of cycles, use a lower CLOCK_FREQUENCY value when running simulations.

2.3 Display Counter module

The `Rate Divider` module allows us to generate slower pulses, given a faster input clock. These `Enable` pulses from the `RateDivider` are used to drive the `EnableDC` signal on `DisplayCounter`. Recall that an enable signal determines whether a flip flop, register, or counter will change on a clock pulse. Compared to the `Rate Divider`, the `DisplayCounter` is much simpler.

The declaration for `DisplayCounter` is as follows:

```
module DisplayCounter (
    input Clock,
    input Reset,
    input EnableDC,
    output [3:0] CounterValue
);
```

Once again, you are encouraged to test your `DisplayCounter` circuit separately before proceeding. Once that is done, you can now put these parts together to build the final circuit.

2.4 Putting it all together

Hopefully, you have already tested your `Rate Divider` and `Display Counter` modules separately. If so, you can put them together in the `part2` module, test it and then submit it for marking.

2.5 Running your design on FPGA

To test your design on an FPGA, you need to make some changes to your code. First, to make it easier to see the numbers counting from 0 to F, you should instantiate a `HEX Decoder` module. You can re-use the `HEX Decoder` you wrote for Lab 2. Simply connect the `CounterValue` output from the `DisplayCounter` module to the input of the `HEX Decoder`.

Also, keep in mind that the FPGA uses a 50 MHz clock.
Use CLOCK_FREQUENCY = 50000000 for the demo.

Once you have made these changes, you can use the mapping shown in Table 2:

module Port Name	Direction	Pin Name
`Clock`	Input	CLOCK_50
`Reset`	Input	SW[9]
`Speed`	Input	SW[1:0]
`CounterValue`	Output	HEX0

Table 2: Module port mapping to DE1-SoC/DE10-Lite pin names

3 Part III

In this part, you need to 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 are represented as:

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

The input to your circuit is one of the eight letters shown in the table above. You must then output the appropriate Morse code for that letter to the output `DotDashOut` using short and long pulses. Short (0.5s) pulses represent dots and long (1.5s) pulses represent dashes. The time between pulses is 0.5 seconds. Similar to Part 2, you must determine how many cycles to count for 0.5 seconds, for a given `CLOCK_FREQUENCY`. We apply the same constraints here, **the automarker may test the project at 100Hz, 500Hz, or 2000Hz**, and will supply this parameter to say which frequency it is testing at. You should use, 50000000Hz if you run the code on the board. Your code should account for this parameter.

You should encode the pattern for each letter using a sequence of 1's and 0's, to correspond to on and off. Since the minimum time is 0.5 seconds, set 0s and 1s 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.

First, write out the exact sequence of 0s and 1s corresponding to each of the letters you need to display. You will see that the sequences have different lengths. For simplicity and to be consistent with what the automarker expects, you should store all the letters using 12-bits. For example, the pattern for A would be stored as `101110000000`.

The top-level module for this part should have the following declaration:

```
module part3
#(parameter CLOCK_FREQUENCY=50000000)(
    input wire ClockIn,
    input wire Reset,
    input wire Start,
    input wire [2:0] Letter,
    output wire DotDashOut,
    output wire NewBitOut
);
```

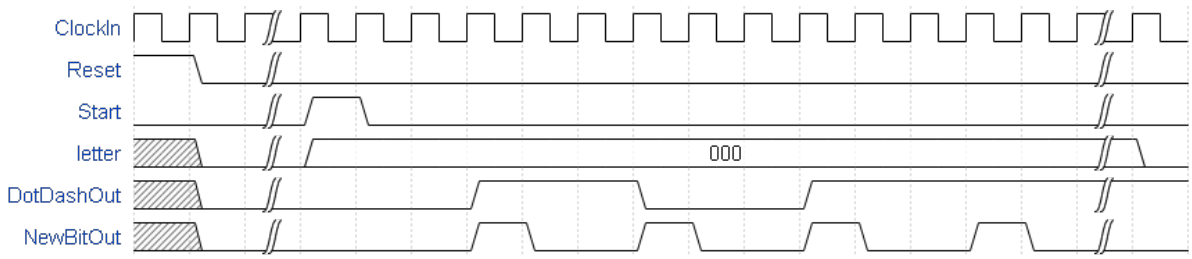

Figure 3: Timing diagram for Part 3

Figure 3 shows a timing diagram of how your circuit should operate, for the letter *A*. 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 asserted, for 1 clock period. Figure 3 shows the first 4 bits of *A* being flashed, for a very low clock frequency. Using the clock in the marker or the FPGA will cause the `DotDashOut` signal to stay 0 or 1 for many more clock cycles.

To make sure your design meets the requirements of the marker, please note the following:

1. There may be many cycles between `Reset` being de-asserted and `Start` being asserted. This is shown with the two wavy lines in Figure 3.
2. Note the 0.5s delay between when `Start` is asserted and when `DotDashOut` starts displaying the sequence.
3. You must assert `NewBitOut` to 1 for 1 clock period for all 12 bits of the sequence, whenever a new DotDashOut is sent, regardless of how many dots/dashes the letter needs. For A `101110000000` we would, each 0.5 seconds output a `DotDashOut` of *1,0,1,1,1,0,0,0,0,0,0,0* over a 6-second period, and NewBitOut, would flash '1' once every 0.5 seconds for those 6-seconds, the first flash being 0.5 seconds after the start signal. This is to ensure that the marker tests your outputs at the right cycles.
4. You can assume that the `Letter` input is held constant while you flash the entire sequence.

With the requirements of the design covered, let's look at how you should design your circuit.

3.1 Designing the encoder

Implementing the `Morse Code` encoder will use concepts you learned earlier such as the `Rate divider` and `Shift registers`, which you used in the previous lab. Since you need to read each bit out, one at a time, you should use a `shift register` to load the letter to flash. When a letter is selected, use the `parallel load` feature of the shift register to load the pattern for that letter to the shift register. Then read each 0 or 1 individually out of a shift register at 0.5 seconds per read.

Similar to supporting different `Speed` values in Part 2, you should use a mux to select the letter to flash, based on the `Letter` input. You should use counters to count the cycles that the `DotDashOut` signal should be high for. Lastly, you should create a pulse, similar to Part 2, for `NewBitOut`.

3.2 Running on FPGA

To run on an FPGA, use the mapping shown in Table 3. Similar to part 2, make sure to update the `CLOCK_FREQUENCY` parameter since the FPGA uses a 50 MHz clock. We highly recommend to test your code on a real FPGA for part 3.

module Port Name	Direction	Pin Name
`ClockIn`	Input	CLOCK_50
`Reset`	Input	SW[9]
`Start`	Input	KEY[0]
`Letter`	Input	SW[2:0]
`DotDashOut`	Output	LEDR[0]
`NewBitOut`	Output	LEDR[1]

Table 3: Module port mapping to DE1-SoC/DE10-Lite pin names

4 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 the instructions prior to submitting your code.

4.1 Part I

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

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

4.2 Part II

For Part II, you need to submit a file named `part2.v` with the following module in it, which takes as a parameter *CLOCK_FREQUENCY*:

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

4.3 Part III

For Part III, you need to submit a file named `part3.v` with the following module in it, which takes as a parameter *CLOCK_FREQUENCY*:

```
module part3(ClockIn, Reset, Start, Letter, DotDashOut, NewBitOut);
```