Part 1. Designing Clock Divider Modules

Clock Dividers have multiple purposes in FPGA design. For the purposes of this lab, there are two specific reasons to use them. The first is that the onboard clock provided by the ZYBO is too fast for the processes in our design, running at 125MHz. A Clock Divider can be used to slow down an input clock by setting a variable that increments with each real clock hit. Once the variable reaches a value specified by the user, the Clock Divider outputs a 1.

For example: An input clock of 10Hz, meaning that the clock strikes 10 times every second, is input into a clock divider. The output of the clock divider is a 2Hz clock. In order to output a 2Hz clock, 2 strikes per second, first you must determine how many strikes of the real clock will occur for each strike of the divided clock. This is done simply by dividing the input clock Hz by the desired output clock Hz. (10/2 = 5)

The value of '5' is what will be set as the value that the internal variable must reach before outputting a 1. By setting a variable which increments by 1 each clock strike, the clock divider will now only output a '1' after 5 actual clock strikes, producing our desired 2Hz clock. Figure 1 shows the output of the same module for an input clock at 1Hz. The corresponding output clock is raised high after 5 ticks of the 1Hz clock, or once every 5 seconds. Note that the output clock frequency is directly related to the input clock frequency. The same concept should be applied when taking the ZYBO's input 125MHz clock and bringing it down to the desired frequencies.

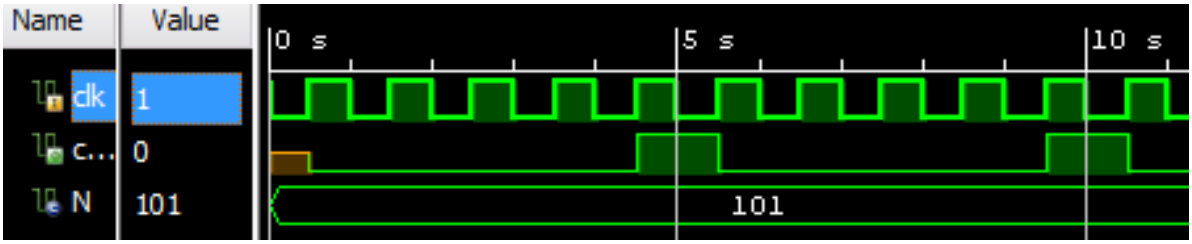

Figure 1. 1Hz to 0.2Hz Clock Divider. N is set at 5, the value which the internal variable must count up to before outputting a 1.

The second reason we will be using clock dividers in this lab is because we need several different clock speeds. As shown in the Figure 1 of the lab description, the Lab 3 design uses the Clock Divider for three different components, each of which requires a different clock speed. This is where generics will be useful, allowing us to use the same clock divider component and set different output clock speeds for both of the counter components, the *toggler*, as well as the *PoV_SS_Driver* component.

Follow the steps below when designing your clock divider.

Step 1-1: Create a clock divider module. Your clock divider should only have a single input and a single output, as well as a generic statement.

Step 1-2: For the default value of N in your generic statement, use what was discussed above to determine what value N should be to bring an input clock of 125MHz down to 2Hz. This will be the speed your counter will increment and decrement at.

Step 1-3: Inside the architecture of our clock divider will be a simple process that includes the input CLK in its sensitivity list. This tells the hardware that this process is dependent on a change in that signal, in this case a change in the *clk* input.

Step 1-4: After the process declaration, your clock divider will need a variable that can be incremented on every clock strike. To allow for this, the variable type should be made *natural* rather than the usual *std_logic*. This will tell your design to treat this variable like an integer. This will allow you to use a statement like:

```
count := count + 1;
```

However, you only want your variable to increment on the rising of the clock. There are a few ways to state this, but for the purposes of this lab you will use:

```
if (clk'EVENT) and (clk = '1') then
```

With this statement, the rest of the If block will only start when a change in *clk* is detected (clk'EVENT), and when the current value is 1. The 'EVENT is an attribute of the signal. Another way to do this would be to use a rising edge statement.

Step 1-5: With your variable incrementing every clock strike, the goal now is to have the output of your clock divider only equal '1' when that variable has reached the N value you set, otherwise it should equal '0'. This is done with another small If block.

**Remember to reset your counting variable once it reaches N, otherwise your clock will output 1 after the desired time, and then stay at one indefinitely.*

Part 2. Designing the *Persistence of Vision Driver*

In lab 2, only one of the digits of the two digit, 7-Segment Display was used. In this lab both digits will be used. Looking at the display pins, you should notice that there are only enough inputs to light up seven segments at any one moment, so how do we light both digits up with separate numbers?

In the previous lab you were only concerned with the seven pins associated with each of the segments on the display. The eighth pin input, the *cathode*, was left unassigned. If you were to instead set that pin to high, the digit on the left would display the number instead of the digit on the right. This is because the display is designed to light one digit or the other depending on what the value of the *cathode* input is.

This is where the concept of *Persistence of Vision* comes into play. *Persistence of Vision* refers to the technique where a light is switched on and off at a fast enough frequency that it looks like the light never turned off at all, as your brain doesn't have enough time to process that the light has turned off before it is on again. This can be applied to our 7-Segment display. By switching the cathode off and on at a fast enough frequency, both digits will appear to be lit consistently at the same time.

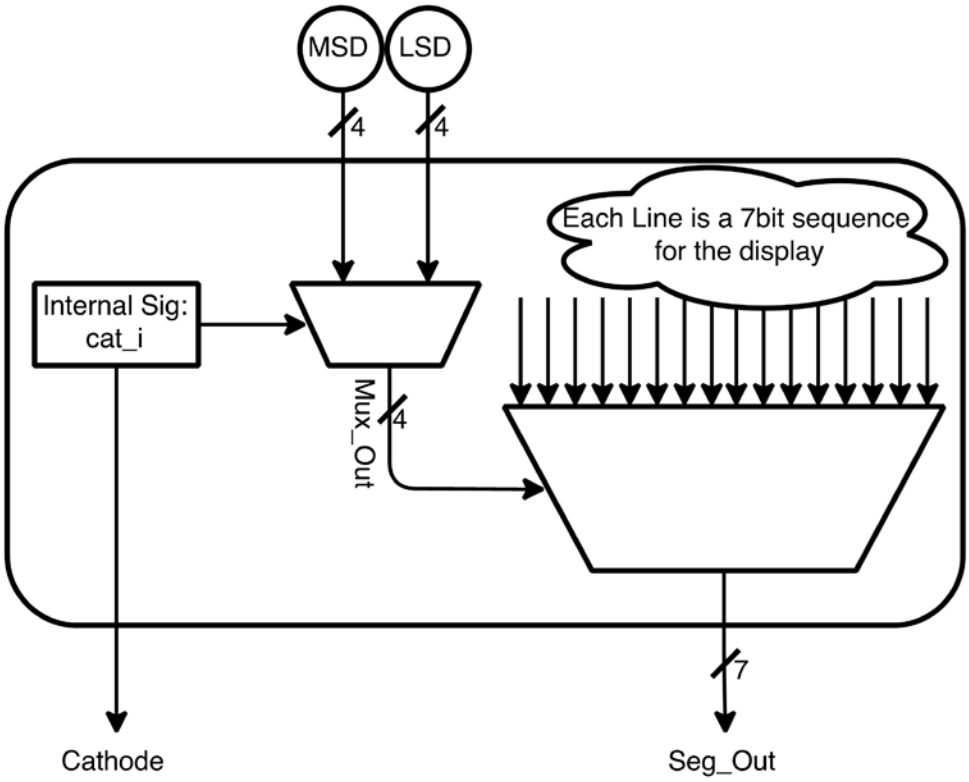

Figure 2. Internal Diagram of the SS_PoV_Driver

Step 2-1: In your *SS_PoV_Driver* you will apply the PoV by using a simple process that switches rapidly between two *cathode* values. Your PoV driver should have **three inputs**, one for your clock, and two for the output values of your two BCD Counters. You will also need **two outputs**, i.e., an output for the seven segment signal, as well as the *cathode.*

Step 2-2: Create an internal signal that will act as the *cathode* signal during the switching process. This signal will invert with each clock strike, and will acts as both the select for the multiplexer that determines which digit to output, as well as provide the cathode output itself. The output of this first multiplexer will then act as the select of the second multiplexer, which determines, based on the value of the digit to display, will the 7bit sequence that needs to be sent to the SS Display.

Step 2-3: Your first multiplexer will use the value of the internal *cathode* signal to select which digit should be displayed, either the MSD or LSD.

Step 2-4: The output of this first multiplexer will then become the select line for the second multiplexer. This one takes the value of the select line and outputs the 7bit sequence needed to actually display the number on the SS Display.

Step 2-5: It's important that the internal *cathode* signal that acts as your select line for the first multiplexer, is also the value sent to the cathode output. The *cathode* output is what determines which of the two displays the current value is displayed on. If you don't use the same signal for the cathode output and first multiplexer select, you may have a glitch where the MSD and LSD are switched on the SS Display.

Expected function: If the current value of the internal *cathode* signal is '0', the output of the first mux should take the value of the LSD BCD counter input and use it as the input of the second multiplexer. Depending on that value, the 7-segment digit on the right of the display should show that value.

Part 3. Top Level Design

Step 3-1: You have been provided a BCD counter module, and a toggle module. Create a *Lab3.vhd* file that instantiates and interconnects two BCD counters and two toggles, three clock division modules, and a PoV 7-Segment Driver.

Step 3-2: Write a testbench for the Lab3 design. Verify that it works as intended before proceeding. Simulate with the test vectors shown in *Lab Work* part 4. Save the resulting waveform and truth table and include them in your lab report.

Step 3-3: Use Vivado's I/O Planning tool to design your constraints file. The ZYBO board helpfully provides an onboard clock for use in projects that runs at 125MHz. To use this clock, simple assign your clock input to *Pin L16* in your constraint file.

Step 3-4: The rest of your constraints should match the assignments listed in Table 1 below.

PMOD Switches	Onboard Switches	Onboard Buttons
SW1 = B(3)	SW1 = A(3)	BTN3 = Enable
SW2 = B(2)	SW1 = A(2)	BTN1 = Load
SW3 = B(1)	SW1 = A(1)	BTN0 = Up_Dn
SW4 = B(0)	SW1 = A(0)	BTN2 = Reset

Step 3-5: Your Two-Digit 7-Segment Display should be mapped at it was in lab 2, however this time, P5 of the J2 connector should be assigned as the *Cathode* output.