# Laboratory Exercise #9
# Counters, Clock Dividers, and Debounce Circuits

ECEN 248: Introduction to Digital Design

Department of Electrical and Computer Engineering
Texas A&M University

# 1 Introduction

The primary objective of this week's lab assignment is to reinforce your knowledge of sequential circuits by introducing an important synchronous sequential circuit, the binary counter. The lab will guide you through the design of a binary up-counter using familiar combinational components as well as sequential components discussed in the previous lab. Furthermore, the lab will demonstrate two important use cases for binary counters, namely clock frequency division and I/O debouncing. Due to the practical nature of this lab assignment, you will be testing all of your designs on the FPGA board.

# 2 Background

The following subsection will review the theory necessary for completion of this lab assignment. The pre-lab will test your knowledge of these concepts.

## 2.1 Counters

With the components you have been introduced to at this point, you can construct a wide variety of useful synchronous circuits. One particularly important circuit we will discuss in this lab is the binary counter. A simple binary up-counter constructed from half-adders and flip-flops is depicted in Figure 1. It is interesting to note that the half-adders are chained together to create a ripple-carry adder that increments the binary value stored in the flip-flops by 1 on every rising-edge of the clock if the enable ($En$) is high. If $En$ is low, the binary value stored in the counter does not change. Notice the *Reset* signal, which connects to each flip-flop in Figure 1. We have not discussed *Reset* signals yet; however, the idea is quite simple. For certain synchronous circuits, the beginning state is important. *Reset* provides a mechanism to initialize the state of a synchronous circuit. When the *Reset* signal is asserted, the flip-flops will *reset* to a known state. Typically, the *reset* state will be '0'; however, it is not uncommon to *reset* a flip-flop to '1'. Just like the clock signal, the *Reset* signal is a global net distributed to all synchronous components.

Many use cases exist for the binary counter such as input de-bouncing and clock frequency division. The next subsections will expound on those topics, while the experiments in this lab will provide you with experience working with these sorts of circuits.

## 2.2 Clock Division

Clock frequency division refers to the process of generating a slower clock from a faster clock. For example, supposed we have two synchronous circuits driven by separate clocks, $Clk_1$ and $Clk_2$. If the synchronous circuit driven by $Clk_2$ is slower than the one driven by $Clk_1$, then we could use a clock divider circuit to

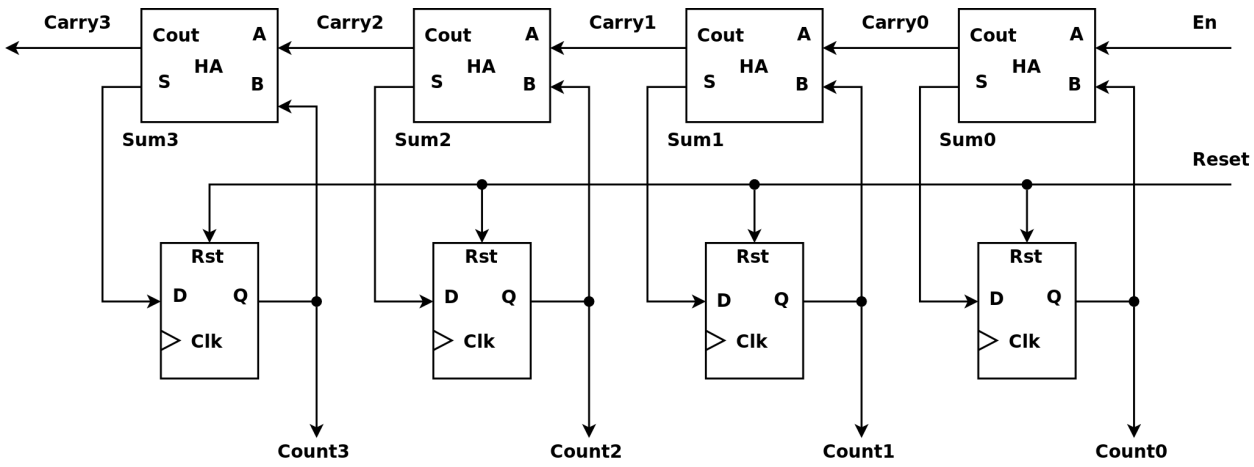


Figure 1: 4-bit Binary Up-Counter

generate the slow clock with the fast clock. Then both circuits can essentially be driven off of the same clock, eliminating the need for two different oscillators. Figure 2 illustrates this concept. If the frequency of the fast clock, $f_1$, must be divided by a power-of-2 in order to generate the slow clock, $f_2$, then a simple binary counter can be used. If $f_2 = \frac{f1}{2^n}$, then an $n$-bit binary counter will be sufficient. We can convince ourselves that this is true by observing the fact that when counting in binary, the least significant bit toggles twice as fast as the next significant bit. The same is true of the next significant bit and so forth.

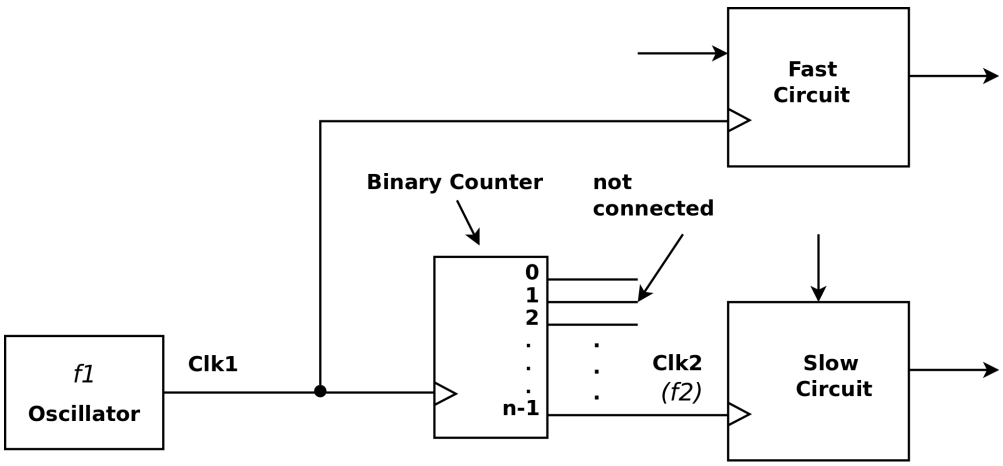


Figure 2: Clock Frequency Division Using A Binary Counter

## 2.3 Signal Debouncing

Up to this point, we have assumed that switches and push-buttons, used for receiving input from the user, are idea in terms of the output signals that they produce. Unfortunately, this is not the case in the real world because these devices rely on less than ideal electromechanical contacts to generate electrical signals from mechanical events. Electrical chatter as seen in Figure 3 is generated every time a switch is flipped or a button is pressed. This chatter (a.k.a switch bounce) consists of several short duration pulses, which appear as a bouncing signal in the interval immediately after the electromechanical contacts are brought together.

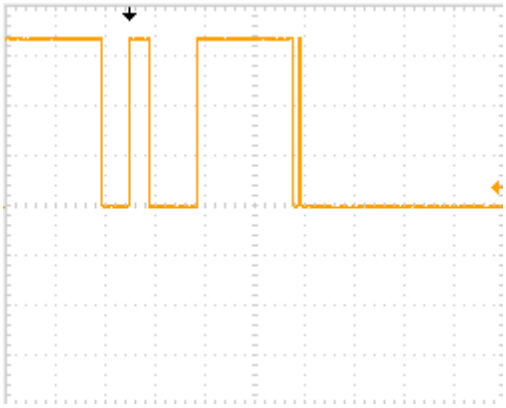

Figure 3: Electrical Chatter

Oftentimes synchronous circuits can be sensitive to this electrical chatter, in which case some additional circuitry must be added to the design to eliminate the bounce in the signal. Many techniques exist for debouncing a signal, but for certain situations, a simple counter circuit will suffice.

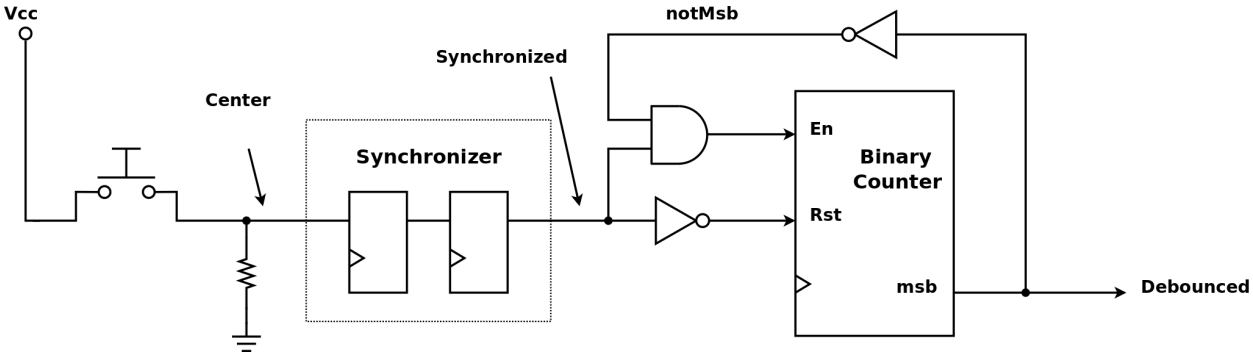


Figure 4: Simple Debounce Circuit

The diagram in Figure 4 demonstrates how a counter can be used to debounce a signal from a push-button. For the moment, ignore the *synchronizer* portion and examine the right half of the debounce circuit. While the *Synchronized* signal is LOW, the counter is held in *reset* (i.e. $Count = 0$). Likewise, the most-significant-bit (msb) of the counter is LOW; therefore, *notMsb* is HIGH. When the *Synchronized* signal transitions to HIGH, the *reset* of the counter is released so the counter will begin counting. Figure 5 illustrates this behavior. Assuming the bit width of the counter is sufficiently sized, the msb of the counter will never become HIGH while the *Synchronized* signal bounces because the counter will continually be *reset*. Thus, the *Debounced* signal should remain LOW during this period. Once the *Synchronized* signal smooths out, the counter will *saturate*. In other words, the msb of the counter will become HIGH, which will in turn force *En* to go LOW, stopping the counter from counting. At this point, the *Debounced* signal will be HIGH until the *Synchronized* signal goes LOW again, in which case the process repeats.

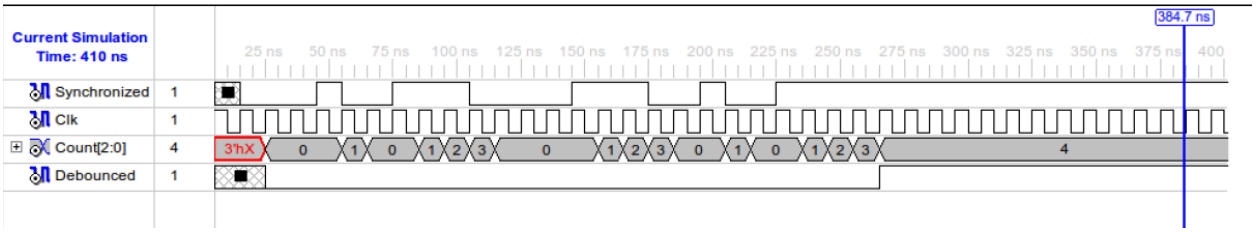


Figure 5: Debounce Circuit Waveform

We deferred the discussion concerning the *Synchronizer* circuit until the end because it is mostly of practical matters and rarely discussed in theory. The signal created by the push-button is not only bouncy but also asynchronous. In other words, the signal could rise or fall at any point in time regardless of the clock signal. This can be problematic in a synchronous circuit because the input to a flip-flop must be stable for a small window of time before (*setup-time*) and after (*hold-time*) the active edge of the clock. For a rising-edge triggered flip-flop, the active edge is the rising-edge. If the *setup* or *hold-time* of a flip-flop is violated, the flip-flop will remain in a metastable state (i.e. not really '1' or '0') for one or two clock cycles. The probability of the flip-flop remaining metastable falls off rapidly as time goes on so for most applications, a *synchronizer* circuit can be created with two cascaded flip-flops. Simply put, the *synchronizer* circuit serves to align the signal from the button with the active edge of the clock. This ensures that by the time the button signal propagates into the rest of the synchronous circuit, it is not violating *setup/hold time* and will not cause the flip-flops after the synchronizer to become metastable.

The remaining components in Figure 4 should look familiar. For example, the pull-down resistor brings the output of the push-button circuit to a LOW state when the button is not pressed. This configuration is similar to that of the switches from earlier labs. For the experiments in lab, we will use the push-buttons on the ZYBO board and the XDC to specify the use of the pull-down resistor within the FPGA.

# 3 Pre-lab

The pre-lab questions below will test you on your knowledge of the material presented above. Please answer the following questions completely:

1. Supposed we want to build a 32-bit counter using the circuit in Figure 1. If the gates used to construct the half-adders in the circuit are assumed to have a 2ns delay each, and the flip-flop overhead is assumed to be negligible, what is the maximum clock frequency we could use to drive our counter? Given the clock frequency you just calculated, how long would it take this counter to roll over (i.e. return to 0). Please show your calculations.

2. Consider the use of a counter to divide an incoming clock. If our incoming clock signal is 32.768 kHz and we need a 64 Hz signal, how many bits would our counter need to have (i.e. what is $n$) to divide the incoming clock correctly.

3. If the Seconds per Division setting for Figure 3 was set to 1 ms/div, what do you think would be a good bit width for the counter in Figure 4, assuming a 50 MHz clock signal was driving the counter? Explain your answer.

# 4 Lab Procedure

The following lab exercises will guide you through the implementation of the circuits discussed in the background section above. For the first two experiments, we will do things differently by introducing the behavioral description before using structural Verilog. The reasons for this should become clear shortly.

## 4.1 Experiment 1

The objective of the first experiment is two-fold. First, we would like to give you the opportunity to describe a synchronous sequential circuit in Verilog and actually load it onto the FPGA board. Second, we would like to provide you with hands on experience dealing with counters and clock frequency division. These are important concepts that you will need to understand when you go to design more complex digital circuits.

**Note:** You will need to pair up with someone in the lab for this experiment because we do not have enough logic analyzers for every workstation. Please take a moment to find a friend in the lab who is willing to work with you. If you do not have a friend, ask the TA to find you one.

1. Invoke Vivado and create a new project called lab9.

2. The following steps will walk you through the implementation of a simple counter circuit that we will use to divide the rate of our system clock.

   (a) Type the Verilog code below into a source file and save it as 'clock_divider.v':

```
1  `timescale 1ns / 1ps
   `default_nettype none
3
   /*This simple module will demonstrate the concept of clock frequency*
5   *division using a simple counter. We will use behavioral Verilog   *
    *for our circuit description.                                       */
7
   module clock_divider(ClkOut, ClkIn);
9
       /*output port needs to be a reg because we will drive it with *
11      *a behavioral statement                                        */
       output wire [3:0] ClkOut;
13     input wire ClkIn; //wires can drive regs

15    /*-this is a keyword we have not seen yet!*
        *-as the name implies, it is a parameter *
17      * that can be changed at compile time... */
       parameter n = 5;//make count 6-bits for now...
19
       reg [n:0] Count; //count bit width is based on n! how cool is that!
21
       /*simple behavioral construct to describe a counter...        *
23      *Are you ready for this?!?                                     */
       always@(posedge ClkIn) //should look familiar...
25         Count <= Count + 1; //yea that's it

27      /*now we need to wire up our ClkOut which is a 4-bit wire*/
        /*Wire up to most-significant bits*/
29      assign ClkOut[3:0] = Count[n:n-3];

31 endmodule //sorry if this was not more exciting
```

(b) Now add the above source file and the 'clock_divider.xdc' found in the course directory to your Vivado project.

(c) Synthesize and Implement the design as outlined in the previous lab.

3. The output of the counter changes far too rapidly for us to view with LEDs so we will need to employ the logic analyzer on our work bench. The logic analyzer we are going to use is integrated in Agilent InfiniiVision 2000 X-Series Oscilloscope. Unlike the oscilloscope, the logic analyzer on your workbench can probe up to 16 digital signals at once. It also has the capability of grouping signals into buses, which can be viewed in decimal or hexadecimal format. The waveforms that you will see on the logic analyzer today have many similarities to the waveforms you have previously seen in simulation. The difference, however, is that the signals you see on the logic analyzer are being created with actual hardware! The following steps will direct you through the process of connecting the logic analyzer to the ZYBO board, as well as the initial setup of the logic analyzer.

   (a) Before connecting the logic analyzer to the output of our counter, ensure that the FPGA board

is turned off. Now, locate the JB connector towards the bottom-right of the FPGA board. Once you have found the JB connector, open the XDC file and compare it to the diagram in Figure 6. You should now be able to locate the *ClkOut* signals of your clock divider.

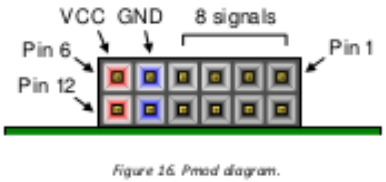


Figure 6: Pinout for the ZYBO Board Pmod courtesy of Xilinx®

(b) One group of wires (a.k.a channels) is plugged into the front of the logic analyzer. Now, examine the end of the cable, taking note of the individual channel labels. On the end of each channel, you should see a 'GND' port and the channel port. Use bread board and jumper wires to connect the all the 'GND' ports on channel 0,1,2,3 to the 'GND' on JB. Now connect channels 0 through 3 to ClkOut[0] through ClkOut[3] on JB in a similar manner. Have the TA inspect your setup before moving on.

(c) Turn the logic analyzer on. See Figure 7.

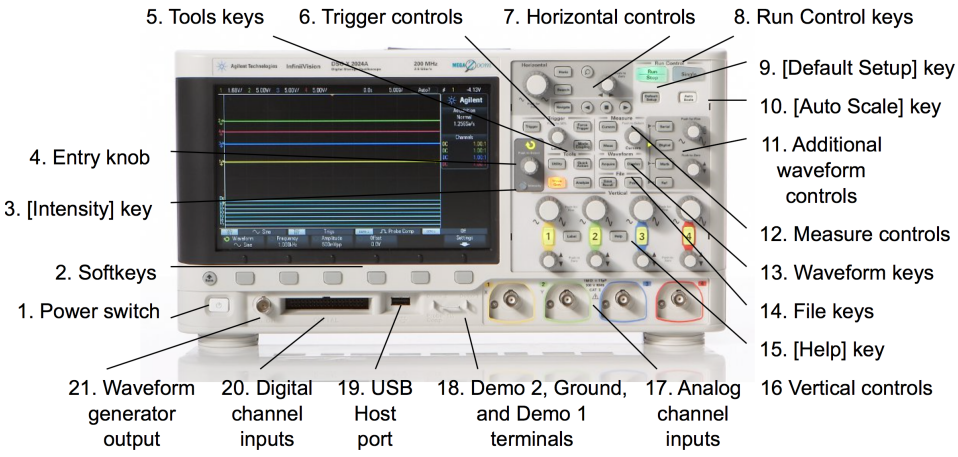


Figure 7: Agilent InfiniiVision 2000 X-Series Oscilloscope Front Panel

(d) Once the Logic Analyzer boots up, press the **Digital** button on the front panel.

(e) Press the **Channel** softkey. Check channels from D0 to D3 and uncheck all the other channels. You can turn the **Entry knob** to choose different channels and press the **Entry knob** to check/uncheck.

(f) Now we would like to assign customized labels to the channels. Press the **Label** button. Then press the **Channel** softkey. Turn the Entry knob to move the highlight cursor to D0:D0 and select it by pressing the Entry knob.

(g) Press **Enter** softkey and then press **Spell** softkey. Use the Entry knob to select and enter COUNT2 one character by one character in the **New label** field. Then press **Apply New Label** softkey to assign channel D0 with a COUNT2 label.

(h) Repeat the same process to assign a COUNT3 label to D1 channel, a COUNT4 label to D2 channel, and a COUNT5 label to D3 channel.

(i) Turn the FPGA board on and program the FPGA with the clock divider circuit. Press **Run** on the logic analyzer once the FPGA has been programmed successfully. If everything is working properly, you should see four separate clock signals on the logic analyzer. Press **Auto Scale** button or use the Horizontal Knob to adjust your horizontal scale.

4. To gain a little more experience working with the logic analyzer, we will first perform some simple time measurements, and then we will group our *ClkOut* signals together to view the counter in operation.

   (a) Press the **Cursor** button. Use the **cursors knob** to select and move the **X1** and **X2** cursors on the screen. Measure the periods of COUNT2, COUNT3, COUNT4, COUNT5. You will see the difference of **X1** and **X2** in terms of time on the right side of the screen as $\Delta$X.

   (b) Press **Digital** Button, then press **Bus** softkey. You will see there are "Bus", "Channel", "Select", "Deselect", "Base" manuals at the bottom of the screen. Press the **Bus** softkey, then check Bus1. Similarly, press **Channel** softkey and select D0, D1, D2, D3. You will see the heximal value of COUNT[5:2] as a bus clearly.

## 4.2 Experiment 2

For Experiment 2, we will use our Verilog skills to design the up-counter discussed in the background section of this manual. We will drive this up-counter with the clock divider created in the previous experiment and wire the output of the up-counter to the LEDs on the FPGA board. For the *En* signal, we will utilize the BTN0(R18) push-button. The steps below will guide you through the experiment.

1. Design the up-counter illustrated in Figure 1.

   (a) Create a source file called 'half_adder.v' and describe the half adder circuit found in Lab 3 using *dataflow* Verilog. You do not need to include delays in your Verilog code. For consistency, use the following module interface:

```
module half_adder(S, Cout, A, B);
```

**Hint:** If you are unsure where to get started, take a look at the source file for the full-adder you designed on Lab 6!

(b) Create a source file called 'up_counter.v' and construct the circuit found in Figure 1 using Verilog. The code snippet below should help you get started.

```
1  `timescale 1ns / 1ps //specify 1ns for each delay unit
   `default_nettype none
3
   /*This module describes a simple 3-bit up-counter using *
5   *half-adder modules built in the previous step         */

7  module up_counter(Count, Carry2, En, Clk, Rst);

9      /*Count output needs to be a reg */
       output reg [2:0] Count;
11     output wire Carry2;
       /*inputs are wires*/
13     input wire En, Clk, Rst;
       /*intermediate nets*/
15     wire [2:0] Carry, Sum;
       /*Let's create and instantiated a wrapper for the 3-bit counter first*/
17     Threebit_counter UC1(Sum,Carry2,Count,En);
       /*Describe positive edge triggered flip-flops for count*/
19     /*Including ``posedge Rst" in the sensitivity list      *
        *implies an asynchronous reset!                       */
21     always@(posedge Clk or posedge Rst)
           if(Rst)//if Rst == 1'b1
23             Count <= 0; //reset count
           else //otherwise, latch sum
25             Count <= Sum;

27 endmodule

29 //Yes, you can have two modules in one file!
   module Threebit_counter(Sum,Carry2,Count, En);
31  // first we declare the variables
    input En;
33  input [2:0]Count;
    output [2:0]Sum;
35  output Carry2;
    wire [2:0] Carry;
37
   /*instantiate and wire up half-adders here*/
39

41 /*wire up carry2*/
```

```
43 endmodule
```

2. Now simulate the up-counter you just designed and observe the output waveform to ensure your counter is working properly.

   (a) Copy the 'up_counter_tb.v' test bench from the course directory and simulate it using Vivado.

   (b) Open up the test bench file and try to understand what is going on. You should see that the test bench produces a *Clk* signal. What is the frequency of that signal?

   (c) You should also see that the test bench holds the counter in *reset* for a specific interval of time. How long is that interval?

   (d) After reset is de-asserted, the test bench holds the enable LOW for some amount of time before allowing the counter to run. How long is this time period?

   (e) From within the waveform window, right-click on 'Count' and select $Radix \rightarrow Hexadecimal$ to change the display format of the 'Count' bus to hexadecimal. See Figure 8.

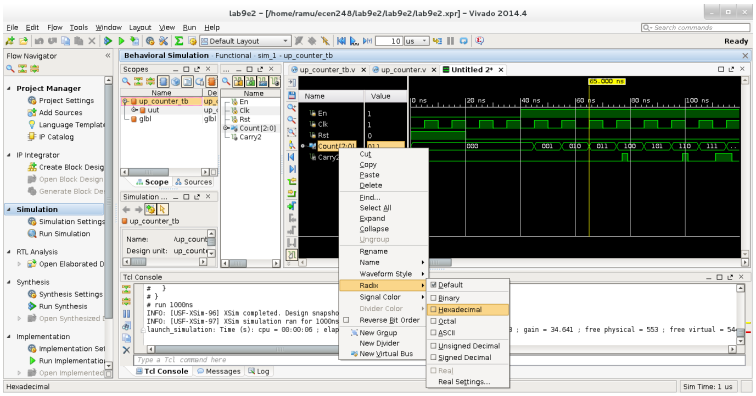


Figure 8: Change Radix to Hexadecimal

   (f) Finally, you should notice that the counter will roll over (i.e. return to 0) after reaching a maximum value. What is this maximum count value and what signal in the waveform could we use to know exactly when the counter is going to roll over?

3. Create a top-level Verilog module that can be used to drive our counter on the FPGA and then load the final design onto the ZYBO board.

   (a) Open the 'clock_divider.v' source file and modify the module such that the *Count* signal is 26 bits wide. This will allow us to divide the clock by $2^{26} = 67,108,864$. If we use a 125MHz clock to drive our frequency divider, what rate will the most significant bit of the divider oscillate at?

(b) Create a source file called 'top_level.v' and use the code snippet below to describe the circuit in Figure 9.

```
1  `timescale 1ns / 1ps //specify 1ns for each delay unit
   `default_nettype none
3
   /*This is the top-level module which wires all
5   *of our synchronous components together. This module *
    *does NOT include synchronizers for the inputs (we   *
7   *will discuss them shortly) so just don't use this in*
    *real application!                                   */
9
   module top_level(LEDs, SWs, BTNs, FastClk);
11
       /*all ports will be wires because we will use
13       structural Verilog to wire everything up*/

15
       /*intermediate nets*/
17     wire [3:0] Clocks;
       reg SlowClk;//will use an always block for mux
19
       /*behavioral description of a mux which
21       selects between the four available clock signals*/
       always@(*) //combinatorial logic
23         case(SWs) //SWs is a 2-bit bus
               2'b00: SlowClk = Clocks[0];//use blocking statement for
25                                        //combinational logic
               //finish this up
27
           endcase
29
       /*instantiate your up_counter here*/
31     /*Hint: if you want to wire a port to just the first 3    *
        *bits of a bus, you can do something like this: LEDs[2:0]*/
33

35     /*instantiate the clock divider. I will do that for you...*/
       clock_divider clk_div0(
37         .ClkOut(Clocks),
           .ClkIn(FastClk)
39     );

41 endmodule
```

(c) Create a file called 'top_level.xdc' and save it in your lab9 directory. Type the XDC starter code shown below. Unfortunately, portions of the XDC are missing. Using the XDCs from the previous labs and Figure 9 as a guide, fix this XDC. You may also find the ZYBO master XDC

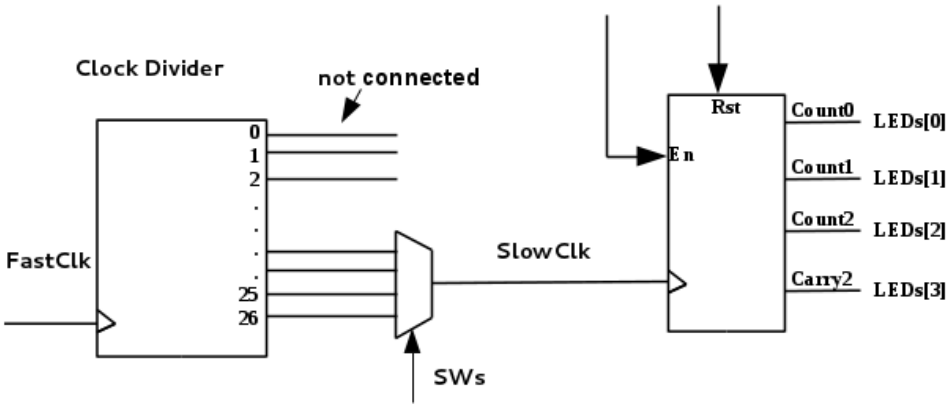


Figure 9: Top-Level Diagram

from Digilent helpful.

```
#Switches
set_property PACKAGE_PIN G15 [get_ports {SWs[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SWs[0]}]
# fill in for SWs[1], connect to SW1

##Buttons
##IO_L20N_T3_34
set_property PACKAGE_PIN R18 [get_ports {BTNs[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {BTNs[0]}]
#fill in for Button 1

##LEDs
##IO_L23P_T3_35
set_property PACKAGE_PIN D18 [get_ports {LEDs[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LEDs[0]}]

#Something is missing here...

set_property PACKAGE_PIN M14 [get_ports {LEDs[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LEDs[3]}]

##Clock signal
##IO_L11P_T1_SRCC_35
set_property PACKAGE_PIN L16 [get_ports FastClk]
set_property IOSTANDARD LVCMOS33 [get_ports  FastClk]
create_clock -add -name sys_clk_pin -period 8.00 -waveform {0 4} [get_ports  FastClk]
```

(d) Add the XDC you just created to your Vivado project and then Synthesize and Implement the design. Correct any errors and then load the design onto the FPGA board.

**Hint:** Errors generated during the synthesis process will be due to issues introduced in your Verilog code, while errors reported during the Implementation phase will be related to the XDC.

(e) Before continuing, it is important that we take note of the warnings that have been generated during the Implementation of this design. For demonstration purposes, we are violating 'good design practice' for FPGA development by create a clock signal from combinational logic. However, the clock signal that we are creating is *very* slow relative to the speed at which the FPGA fabric is able to operate. Thus, our circuit will function as expected. For faster clocks, we should utilize the built-in clock-dividers available on the FPGA.

(f) Experiment with the various combinations of SW1 and SW0, while pressing the 'BTN0' push-button. Observe the rate at which the LEDs toggle. Make note in your lab write-up how the switches affect the LED outputs.

(g) Additionally, press the 'BTN1' push-button and note the result in your lab write-up. Once the design has been found to work, demonstrate your progress to the TA.

### 4.3 Experiment 3

The purpose of the last experiment is to demonstrate the effects of button bounce on an example synchronous circuit. Additionally, we will learn how to eliminate button bounce with the debounce circuit discussed in the background section of this manual. We will begin by mimicing the button bounce signal in verilog testbench. We will then connect that signal to the input of a simple synchronous circuit (as you may have guess, it will be yet another counter!) without proper debouncing. Finally, we will add signal debouncing and compare the results. The following steps will guide you through the experiment.

1. Add 'Bounce_tb.v' as simulation sources from the course directory to your Vivado project. Spend some time to check the codes inside 'Bounce_tb.v'. *Bounced_BTN* is the output signal of a virtual button which mimics the output of a push-button with bounce in the real world. *Is_BTN_Pressed* is a supportive signal. When *Is_BTN_Pressed* is high, it means the virtual button is pressed.

   (a) Figure 10 provides a simple synchronous circuit that will be sensitive to switch bounce. This circuit contains a binary counter with an enable controlled by the output of an edge-detector. The edge-detection circuitry serves two purposes in this circuit. First of all, it synchronizes the asynchronous input from the virtual push-button in the testbench. Second of all, it detects when that signal transitions from LOW to HIGH. Thus, it detects a positive-edge. The output of the edge-detector is a pulse that immediately follows the rising-edge of *Bounced_BTN* and is one clock cycle wide. The designer intended to create a circuit that will increment a counter by one

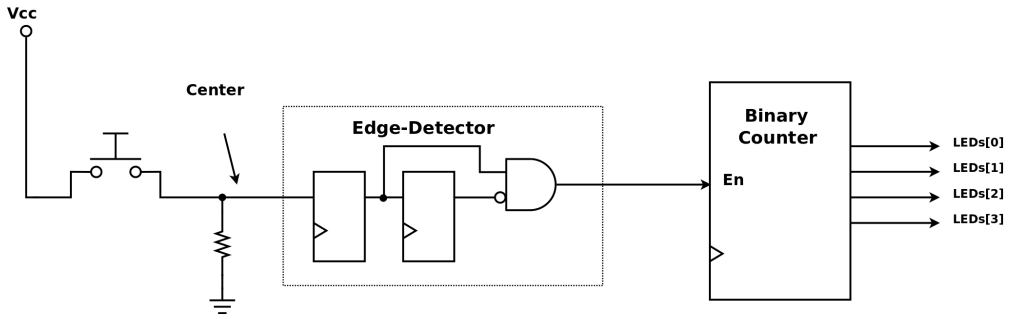


Figure 10: Example Synchronous Circuit without Debounce Circuitry

every time the virtual button with an ouput of *Bounced_BTN* is pressed. Let us see if that is what actually happens.

(b) The circuit in Figure 10 has been implemented for you in 'noDebounce.v'. Copy this file from the course directory to your lab9 directory. Click 'Simulation Settings' on the left panel of Vivado, and click 'simulation' tag. Set the xim.simulate.runtime to be 3000ns. Use 'Bounce_tb.v' as the testbench and run the simulation of 'noDebounce.v'. Check the output waveforms. Does the circuit in 'noDebounce.v' work as expected? Why or Why not?

2. Now that we have seen first hand how switch bounce can affect a synchronous circuit, we should look into how to correct for it.

   (a) Add 'withDebounce.v' from the course directory to your Vivado project. Still use 'Bounce_tb.v' as testbench, but this time, modify the name of uut in 'Bounce_tb.v' so that 'withDebounce.v' is being tested instead of 'noDebounce.v'. Run the simulation of 'withDebounce.v'. Check the output waveforms. Does the counter in 'withDebounce.v' work as expected? Why or Why not?

   (b) Study the circuit in Figure 4. Now take a moment to examine the uncommented Verilog code in 'withDebounce.v'. You should find that this source file contains the circuit in Figure 10 and the circuit in Figure 4 combined into one. Please add comments to the source code and include it in your lab write-up.

   (c) Explain in your lab write-up the operation of the circuit described in 'withDebounce.v'. A block diagram might help with the explanation but is not required.

## 5 Post-lab Deliverables

Please include the following items in your post-lab write-up in addition to the deliverables mentioned in the *Policies and Procedures* document.

(a) Include the source code with comments for **all** modules in lab. You do **not** have to include test bench code. Code without comments will not be accepted!

(b) Include any XDCs that you wrote or modified.

(c) Include screenshots of all waveforms captured during simulation in addition to the test bench console output for each test bench simulation.

(d) Answer all questions throughout the lab manual.

# 6 Important Student Feedback

The last part of the lab requests your feedback. We are continually trying to improve the laboratory exercises to enhance your learning experience, and we are unable to do so without your feedback. Please include the following post-lab deliverables in your lab write-up.

**Note:** If you have any other comments regarding the lab that you wish to bring to your instructor's attention, please feel free to include them as well.

(a) What did you like most about the lab assignment and why? What did you like least about it and why?

(b) Were there any sections of the lab manual that were unclear? If so, what was unclear? Do you have any suggestions for improving the clarity?

(c) What suggestions do you have to improve the overall lab assignment?