ECEN 248 - Lab Report

Lab Number: 8

Lab Title: Counters, Clock Dividers, and Debounce Circuits

Section Number: 513

Student's Name: Abhishek Bhattacharyya

Student's UIN: 731002289

Date: 4/12/2023

TA: Hesam Mazaheri

Objectives:

This week's lab project is to gain more experience of sequential circuits by introducing the idea of a binary counter, a major part of a synchronous sequential circuit. This lab explores how to create a binary up-counter using the combinational and sequential components described in previous labs. The lab will also demonstrate the real-world uses of binary counters, such as clock frequency division and I/O debouncing, and it will give you the opportunity to test designs first-hand on the FPGA board.

Design:

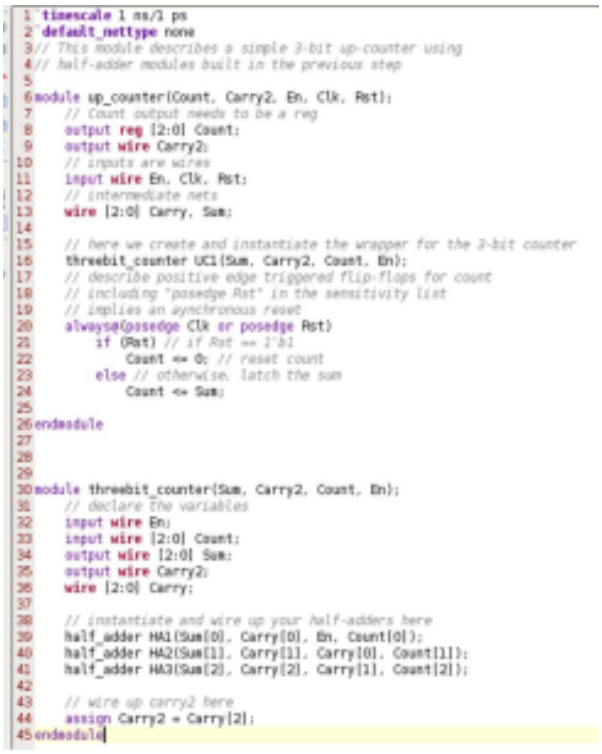
All Verilog code used in this lab is included below.

```
1 `timescale 1ns/1ps
2 `default_nettype none
3
4 // This simple module will demonstrate the concept of clock frequency
5 // division using a simple counter. We will use behavioral Verilog
6 // for our circuit description
7
8 module clock_divider(ClkOut, ClkIn);
9
10     // output port needs to be a reg because we will drive it with
11     // a behavioral statement
12     output wire [3:0] ClkOut;
13     input wire ClkIn; // wires can drive regs
14
15     // this is a keyword we have not seen yet
16     // as the name implies, it is a parameter
17     // that can be changed at compile time
18
19     parameter n = 5; // make count 6-bits for now...
20
21     reg [n:0] Count; // count bit width is based on n!
22
23     // simple heavioral construct to describe a counter
24     always@ (posedge ClkIn)
25         Count <= Count + 1;
26
27     // now we need to wire up our ClkOut which is a 4-bit wire
28     // Wire up to most-significant bits
29     assign ClkOut[3:0] = Count[n:n-3];
30
31     initial
32         begin
33             Count <= 0;
34         end
35 endmodule
```

Figure 1: *Source Code for Clock Divider*

```
1 `timescale 1 ns/1 ps
2 `default_nettype none
3 // This is a module for the half adder previously used in Lab 3
4 // however this one is using the dataflow aspect of Verilog
5
6 module half_adder(S, Cout, A, B);
7     input wire A, B;
8     output wire S, Cout;
9
10     assign Cout = A & B;
11     assign S = A ^ B;
12
13 endmodule
```

Figure 2: *Source Code for Half Adder*



```
`timescale 1 ns/1 ps
`default_nettype none
// This module describes a simple 3-bit up-counter using
// half-adder modules built in the previous step

module up_counter(Count, Carry2, En, Clk, Rst);
    // Count output needs to be a reg
    output reg [2:0] Count;
    output wire Carry2;
    // inputs are wires
    input wire En, Clk, Rst;
    // intermediate nets
    wire [2:0] Carry, Sum;

    // here we create and instantiate the wrapper for the 3-bit counter
    threebit_counter UC1(Sum, Carry2, Count, En);
    // describe positive edge triggered flip-flops for count
    // including "posedge Rst" in the sensitivity list
    // implies an asynchronous reset
    always@(posedge Clk or posedge Rst)
        if (Rst) // if Rst == 1'b1
            Count <= 0; // reset count
        else // otherwise, latch the sum
            Count <= Sum;

endmodule


module threebit_counter(Sum, Carry2, Count, En);
    // declare the variables
    input wire En;
    input wire [2:0] Count;
    output wire [2:0] Sum;
    output wire Carry2;
    wire [2:0] Carry;

    // instantiate and wire up your half-adders here
    half_adder HA1(Sum[0], Carry[0], En, Count[0]);
    half_adder HA2(Sum[1], Carry[1], Carry[0], Count[1]);
    half_adder HA3(Sum[2], Carry[2], Carry[1], Count[2]);

    // wire up carry2 here
    assign Carry2 = Carry[2];
endmodule
```

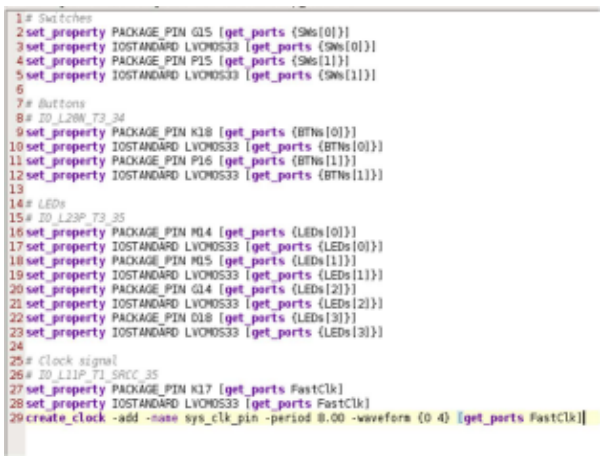
Figure 3: *Source Code for Up-Counter and 3-bit Counter*

```
`timescale 1ns / 1ps // specify 1 ns for each delay unit
`default_nettype none

// this is the top-level module which wires all
// of our synchronous components together. this module
// does NOT include synchronizers for the inputs (we
// will discuss them shortly) so just don't use this in
// a real application

module top_level(LEDs, SWs, BTNs, FastClk);
    // all ports will be wires
    output wire [3:0] LEDs;
    input wire [1:0] SWs;
    input wire [1:0] BTNs;
    input wire FastClk;

    // there are 4 LEDs, 2 switches, and 2 buttons
    // FastClk is a 1-bit wide input

    // intermediate nets
    wire [3:0] Clocks;
    reg SlowClk; // will use an always block for MUX

    // behavioral description of a four-way MUX
    always@(*) // combitional logic
        case(SWs) // SWs is a 2-bit bus
            2'b00: SlowClk = Clocks[0];
            2'b01: SlowClk = Clocks[1];
            2'b10: SlowClk = Clocks[2];
            2'b11: SlowClk = Clocks[3];
            // use blocking statements for
            // combinational logic

        endcase
    // here i instantiated the up counter
    up_counter UCO(LEDs[2:0], LEDs[3], BTNs[0], SlowClk, BTNs[1]);

    // instantiate the clock divider
    clock_divider clk_div0(
        .ClkOut(Clocks),
        .ClkIn(FastClk)
    );
endmodule
```

Figure 4: *Source Code for Top-Level*

```
`timescale 1ns / 1ps
`default_nettype none
module withDebounce(LEDs, BTN, Clk);
    output reg [3:0] LEDs;
    input wire BTN, Clk;
    /*-this is a keyword we have not seen yet!*
     *-as the name implies, it is a parameter *
     * that can be changed at compile time... */
    parameter n = 5;
    wire notMsb, Rst, En, Debounced;
    reg Synchronizer0, Synchronized;
    reg [n-1:0] Count;
    reg edge_detect0;
    wire rising_edge;
    /*This is just for simulation*/
     initial
        LEDs=0;
   /*********************************************/
   /* Debounce circuitry!!!                     */
   /*********************************************/
    always@(posedge Clk)
      begin
        // this sets synchronizer0 to the btn
        Synchronizer0 <= BTN;
        // this sets the output of the synchronizer circuit,
        // synchronized to the synchronizer0 that was set
        Synchronized <= Synchronizer0;
      end
    always@(posedge Clk)
        // if the reset is on, then the count will be less
        // than or equal to 0
        if(Rst)
            Count <= 0;
        // if the enable is on, then the count will be less
        // than or equal to the count plus one
        else if(En)
            Count <= Count + 1;
    // this assigns notMsb with the Count[n-1] being NOT
    assign notMsb = ~Count[n-1];
    // this assigns En with the notMsb that was assigned
    // above and the Synchronized output from the Synchronizer
    // part of the circuit
    assign En = notMsb & Synchronized;
    // this assigns reset with the Synchronized output being NOT
    assign Rst = ~Synchronized;
    // this assigned Debounced with the Count[n-1] that was
    // earlier with the notMsb part of the circuit
    assign Debounced = Count[n-1];
   /*********************************************/
   /* End of Debounce circuitry!!!              */
   /*********************************************/
    always@(posedge Clk)
        edge_detect0 <= Debounced;
    assign rising_edge = ~edge_detect0 & Debounced;
    always@(posedge Clk)
        if(rising_edge)
            LEDs <= LEDs + 1;
endmodule
```

Figure 5: *Source Code for with Debounce*



```
# Switches
set_property PACKAGE_PIN G15 [get_ports {SWs[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SWs[0]}]
set_property PACKAGE_PIN P15 [get_ports {SWs[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SWs[1]}]

# Buttons
# IO_L20N_T3_34
set_property PACKAGE_PIN K18 [get_ports {BTNs[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {BTNs[0]}]
set_property PACKAGE_PIN P16 [get_ports {BTNs[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {BTNs[1]}]

# LEDs
# IO_L23P_T3_35
set_property PACKAGE_PIN M14 [get_ports {LEDs[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LEDs[0]}]
set_property PACKAGE_PIN M15 [get_ports {LEDs[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LEDs[1]}]
set_property PACKAGE_PIN G14 [get_ports {LEDs[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LEDs[2]}]
set_property PACKAGE_PIN D18 [get_ports {LEDs[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LEDs[3]}]

# Clock signal
# IO_L11P_T1_SRCC_35
set_property PACKAGE_PIN K17 [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]
```

Figure 6: *Source Code for Top-Level XDC File*

Results:

After the first two parts of this experiment were completed, I was able to create and analyze the binary counter. By analyzing the oscilloscope output (shown below), the clock period of each counter within the clock divider module was calculated. These results showed that the counter module was working as expected, dividing the clock signal.

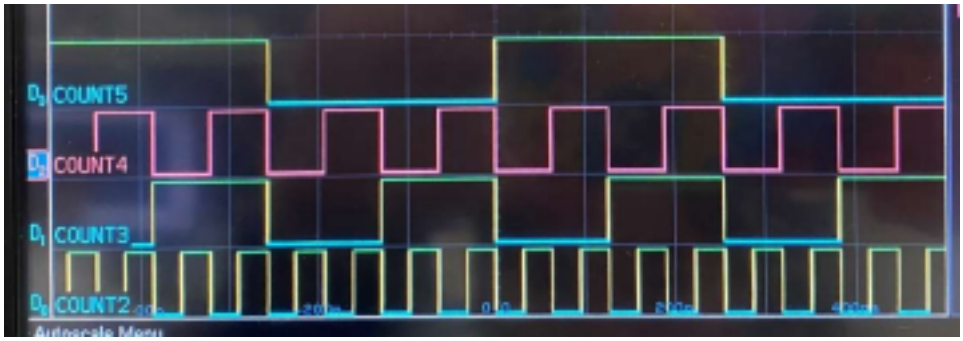


Figure 7: *Oscilloscope Output*

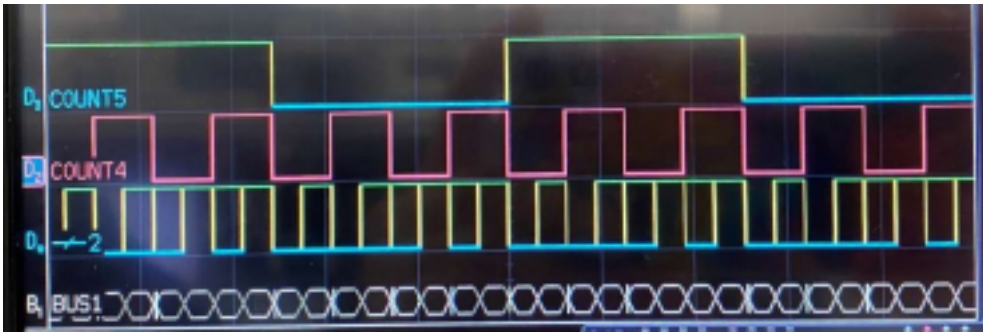


Figure 8: *Oscilloscope Output with Bus*

Conclusion:

Utilizing existing experience working with Varileog and picking up new abilities, including utilizing an oscilloscope, I was able to successfully finish this lab. This lab taught me how to use the oscilloscope to evaluate design modules such as the counter in Verilog using the FPGA board. I also learnt how to develop a debounce circuit to get rid of electrical chatter in a design, as well as how to use the package pin numbers on the ZYBO board to create an XDC file for the top-level design module. The significance of a debounce circuit and how it works were also important lessons I learned from this lab.

Post-Lab Deliverables

1. ***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!***

   This is all included above in the **Design** section.

2. ***Include any XDC files that you wrote or modified.***

   This is all included above in the **Design** section.

3. ***Include screenshots of all waveforms captured during simulation in addition to the test bench console output for each test bench simulation.***

Figure 9: *Waveform for Up-Counter*

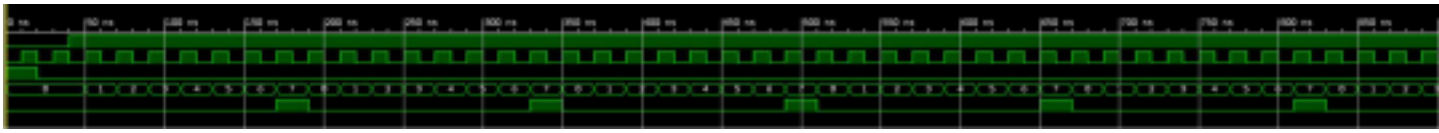


Figure 10: *Waveform for Up-Counter with Hexadecimal Output*

```
# }
# run 1000ns
INFO: [USF-XSim-96] XSim completed. Design snapshot 'up_counter_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
```

Figure 11: *Console Output for Up-Counter and Up-Counter with Hexadecimal Output*

Figure 12: *Waveform for No Debounce*

```
#   }
# }
# run 3000ns
INFO: [USF-XSim-96] XSim completed. Design snapshot 'bounce_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 3000ns
```

Figure 13: *Console Output for No Debounce*

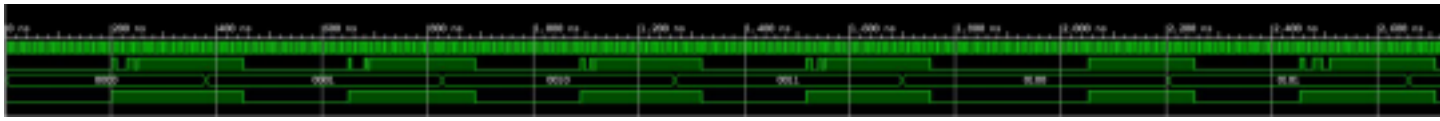


Figure 14: *Waveform for With Debounce*

```
#   }
# }
# run 3000ns
INFO: [USF-XSim-96] XSim completed. Design snapshot 'bounce_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 3000ns
```

Figure 15: *Console Output for With Debounce*

4. ***Answer all questions throughout the lab manual.***
    a. ***Experiment Part 1, 4.a (clock periods and answer the question): How do your count signals differ from the original clock signal?***

COUNT2 = 128.00ns
COUNT3 = 64.00ns
COUNT4 = 512.00ns
COUNT5 = 256.00ns

These are as expected - the counters are uniformly dividing the clock signal in multiples of 2.

    b. ***Experiment Part 2, 2.b: You should see that the test bench produces a Clk signal. What is the frequency of that signal?***

F = 1/Tc; so Freq = 1/10ns = 10000 MHz = 10 BHz

    c. ***Experiment Part 2, 2.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?***

Reset interval = 20 ns.

    d. ***Experiment Part 2, 2.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?***

Time period is 20ns.

e. ***Experiment Part 2, 2.f: Finally, you should notice that the counter will roll over 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?***

Maximum count is 1111 (F in hexadecimal); the counter will roll over when Carry3 is equal to 1.

f. ***Experiment Part 2, 3.a: If we use a 125MHz clock to drive our frequency divider, what rate will the most significant bit of the divider oscillate at?***

Frequency = (125MHz) / 2^{26} = 1.86 Hz

Time = 1/f = 1/1.86Hz = 0.537 s

g. ***Experiment Part 3, 1.b: Does the circuit in 'noDebounce.v' work as expected? Why or why not?***

Yes, the circuit in the module is working as expected. You can see the electrical noise (no debouncing is used) in Figure 12, which is the blank outputs between the valid number outputs. For example, at the output is 0000 there's a space with no output (".") and then it turns into 0001. Without a debounce circuit,we can see the switch bounce within the waveform and this can be fixed by implementing debounce in the design.

h. ***Experiment Part 3, 2.a: Does the counter in 'withDebounce.v' work as expected? Why or why not?***

The "withDebounce.v" module also works as expected which is shown clearly in Figure 14. While there is still electrical noise, it does not affect the output like it did in Figure 12. For example, when the output switches from 0000 to 0001 there is no noise between the two outputs. So the debounce is working well and correctly as intended.

i. ***Experiment Part 3, 2.c: Explain in your lab write-up the operation of the circuit described in 'withDebounce.v'. Use waveforms to compare operation of noDebounce and withDebounce.***

Figure 7 shows the Verilog code for withDebounce.v with comments explaining each line of code. We can see that there are two flip-flop synchronizers operating on the positive edge of the clock. There are also enable and reset inputs that operate on the positive edge of the clock and take 1 and 0 as inputs, respectively. You can also see the AND and NOT gates connected from the output of the synchronizer to the enable and reset inputs of the binary counter. The output of the binary counter (Msb) passes

through the NOT gate and into the AND gate with the synchronizer output. The final output of the debounce circuit is Msb.

Looking at the differences between the waveforms in Figure 12 and Figure 14 (noDebounce vs withDebounce), there is also a clear change in behavior between the two circuits. With noDebounce, we see the signal bounces after the user input, however, with withDebounce, we do not see any signal bounce because of the debounce circuit being implemented.

Important Student Feedback

1. What did you like most about the lab assignment and why? What did you like least about it and why?

What I liked most about this lab assignment was that I was able to keep practicing with Verilog, as I appreciate gaining more much needed experience. I also had fun using the oscilloscope. What I most liked was just the general tedium of having to troubleshoot my code.

2. 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?

The lab manual was clear and concise to understand.

3. What suggestions do you have to improve the overall lab assignment?

I do not have any suggestions to improve the lab assignment.