California State University Long Beach
Computer Engineering and Computer Science Department
CECS 201 – Computer Logic Design I
Lab 7 – Synchronous Sequential Logic: Registers

Objectives:

- To implement and test a sequential design using Vivado SW and FPGA board
- To practice making **hierarchical** design in Verilog
- To learn how to model **synchronous sequential circuits** in Verilog
- To support learning about synchronous sequential logic: Registers

Theory:

This lab assignment will help you put in practice concepts covered in chapter "Synchronous Sequential Logic" particularly sub-sections "Clocks, D flip-flops, and registers", "Load registers" and "Multi-function registers": please make sure to study the corresponding chapter and do corresponding homework assignment in preparation for this lab assignment.

Setup:

Please follow the setup instruction provided in the Lab 1 assignment.

Assignment:

This lab assignment supports learning about design of Synchronous Sequential Circuits: a Basic Register, a Load Register and a Counter. You will need to create a module and a corresponding testbench for the following designs: an 8-bit (basic) register, an 8-bit load register, and a 32-bit counter and a 16-bit 2-to-1 mux. All registers and the mux will be created using behavioral design. Since the FPGA board has only 16 LEDs, you need to display only 16-bith at a time. Therefore, you will need to create a top-level design (top_counter_32bit), using hierarchical modeling, using a 32-bit counter and a 16-bit mux. The mux will help you select either lower 16-bits or upper 16 bits to be displayed using LEDs on the FPGA board. Finally, you will **implement** a top_counter_32bit on the board. Follow the instructions in the Steps section and the video tutorials.

In the conclusion portion of the report, write the expected results and reflect if the simulation shows the expected results. Also add if the FPGA board shows expected output value for each test case. If not, try to figure out why and write this in the report as well.

Steps:

Both Student Virtual Lab (SVL) and Vivado are installed on the computers in the lab. Feel free to install them on your laptop or desktop.

1. If you need to use SVL, connect to SVL, as per instructions on BB

2. Create a folder **lab7** in the **labs** folder for lab 7 assignment

3. Start Vivado 2016.x and create a project, named lab7-v0

4. For each module in this lab populate the top portion with
   // Company: CECS 201 – Fall 2019
   // Engineer: your name

5. Create a new module to implement an 8-bit (basic) register (Figure 1.) with a module name, inputs and outputs as shown in the Figure 2. Keep in mind, that to reset a register, you need to supply 1 to the "reset" input: irrespective of the values of clk or Din, the value of the register Q will be all 0s after "reset" becomes 1 (until "reset" becomes "0"). Also, in the basic register, if "reset" is 0, the data on Din will be loaded **only** on every positive edge of "clk" signal. Please refer to accompanying video tutorials on BB for more details.

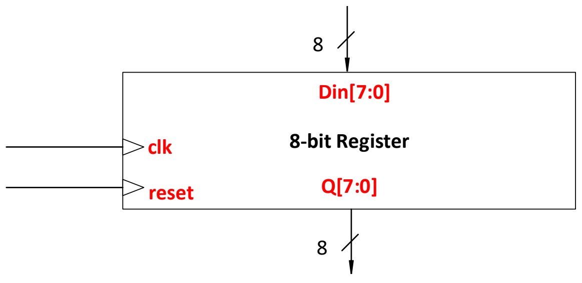
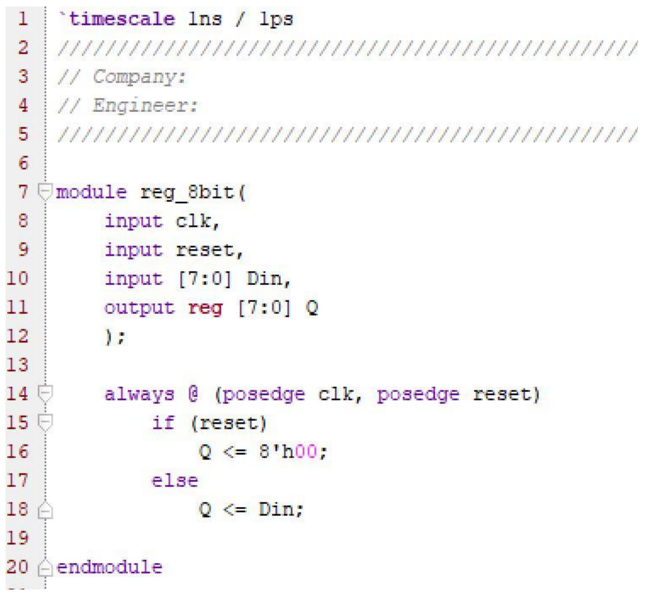


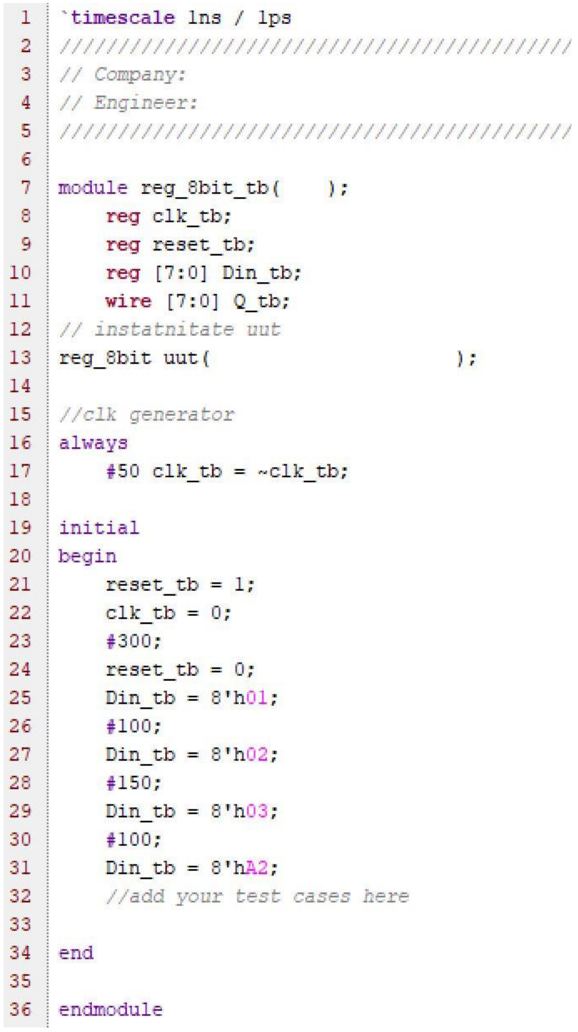
Figure 1. Top level view of an 8-bit register



```
`timescale 1ns / 1ps
//////////////////////////////////////////////////
// Company:
// Engineer:
//////////////////////////////////////////////////

module reg_8bit(
    input clk,
    input reset,
    input [7:0] Din,
    output reg [7:0] Q
    );

    always @ (posedge clk, posedge reset)
        if (reset)
            Q <= 8'h00;
        else
            Q <= Din;

endmodule
```

Figure 2. Module reg_8-bit, its inputs and outputs, and the module body.

6. Create a testbench for 8-bit register as per Figure 3. Instantiate the uut using the variables defined at the beginning of the testbench. Add more tests inside the “initial” statement:
    - o after 100 ns: set reset_tb to 1
    - o after 200 ns: initiate Din_tb with a new value of your choice
    - o after 150 ns: initiate reset_tb to 0 after 150 ns
    - o after 60 ns: initiate another new value of Din



```
`timescale 1ns / 1ps
////////////////////////////////////////////
// Company:
// Engineer:
////////////////////////////////////////////

module reg_8bit_tb(    );
    reg clk_tb;
    reg reset_tb;
    reg [7:0] Din_tb;
    wire [7:0] Q_tb;
// instatnitate uut
reg_8bit uut(                     );

//clk generator
always
    #50 clk_tb = ~clk_tb;

initial
begin
    reset_tb = 1;
    clk_tb = 0;
    #300;
    reset_tb = 0;
    Din_tb = 8'h01;
    #100;
    Din_tb = 8'h02;
    #150;
    Din_tb = 8'h03;
    #100;
    Din_tb = 8'hA2;
    //add your test cases here

end

endmodule
```

Figure 3. Testbench reg_8bit _tb

7. Simulate the testbench for at least 1,400,000 ns and observe the outputs in the simulation window. Include in the report: a table with an expected value at time 1,050,000ns, 1,150,000ns, and 1,250,000ns; and a screenshot of the waveform.

8. Open Schematic (under “RTL Analysis”->” Elaborated Design”) and analyze the output. Include the schematic in the report.

9. Create a new module to implement an 8-bit load register (Figure 4.) with module name, inputs and outputs as shown in the Figure 5. Keep in mind, that to reset a register, you need to supply 1 to the “reset” input: irrespective of value of clk, ld or Din, the value of the register Q will be all 0s after “reset” becomes 1 (until “reset” becomes “0”). Also, if “reset” is 0, only when “ld” is 1, the data on Din will be loaded on every positive edge of “clk” signal. Moreover, if “reset” is 0, only when “ld” is 0, no data will be loaded on every positive edge of “clk” signal. Please refer to accompanying video tutorials on BB for more details.

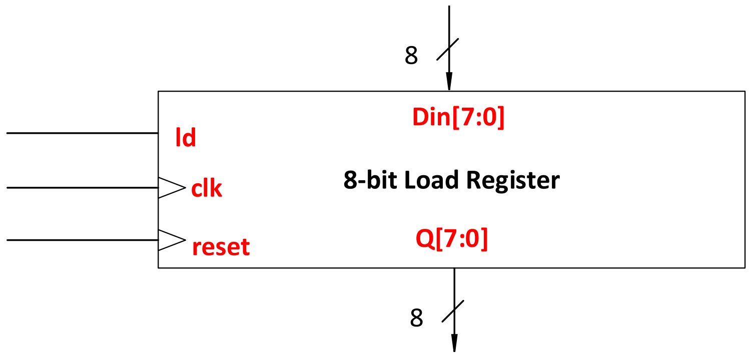
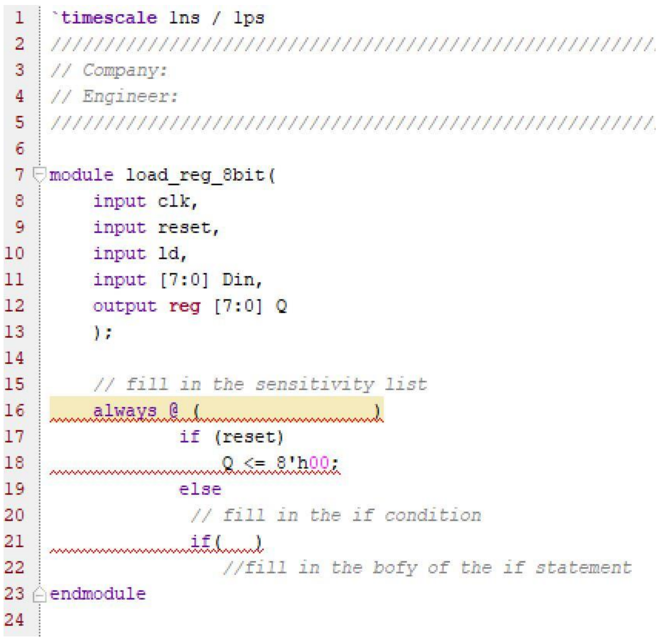


Figure 4. Top level view of an 8-bit load register

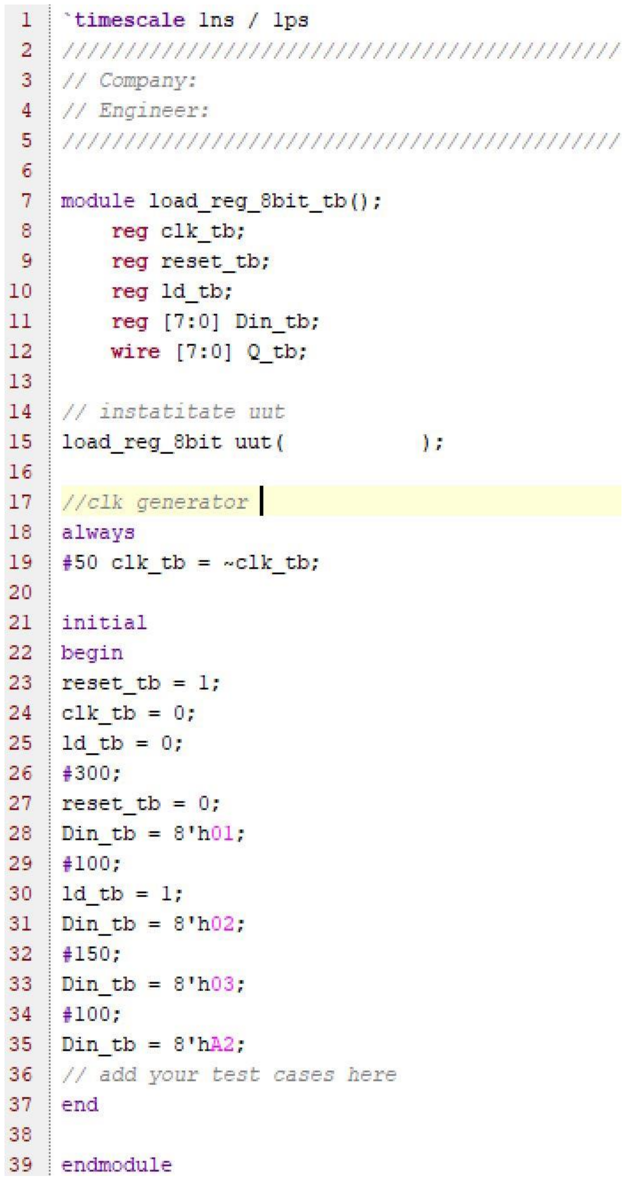


```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////
// Company:
// Engineer:
//////////////////////////////////////////////////////////

module load_reg_8bit(
    input clk,
    input reset,
    input ld,
    input [7:0] Din,
    output reg [7:0] Q
    );

    // fill in the sensitivity list
    always @ (                    )
            if (reset)
                Q <= 8'h00;
            else
             // fill in the if condition
             if(    )
                //fill in the bofy of the if statement
endmodule

```

Figure 5. Module load_reg_8-bit, its inputs and outputs, and the module body.

10. Create a testbench for 8-bit load register as per Figure 6. Instantiate the uut using the variables defined at the beginning of the testbench. Add more tests inside the “initial” statement:

- after 50 ns: initialize both reset_tb and ld_tb to 1 after 50 ns, initiate Din_tb with a new value of your choice
- after another 60 ns: initialize both reset_tb and ld_tb to 0, and initiate another new value of Din
- after 100 ns: initialize reset_tb to 0 and ld_tb to 1



```
`timescale 1ns / 1ps
//////////////////////////////////////////////
// Company:
// Engineer:
//////////////////////////////////////////////

module load_reg_8bit_tb();
    reg clk_tb;
    reg reset_tb;
    reg ld_tb;
    reg [7:0] Din_tb;
    wire [7:0] Q_tb;

// instatitate uut
load_reg_8bit uut(           );

//clk generator
always
#50 clk_tb = ~clk_tb;

initial
begin
reset_tb = 1;
clk_tb = 0;
ld_tb = 0;
#300;
reset_tb = 0;
Din_tb = 8'h01;
#100;
ld_tb = 1;
Din_tb = 8'h02;
#150;
Din_tb = 8'h03;
#100;
Din_tb = 8'hA2;
// add your test cases here
end

endmodule
```

Figure 6. Testbench load_reg_8bit _tb

11. Simulate the testbench for enough time to show all the new test cases and observe the outputs in the simulation window. Include in the report: a table with an expected value at time 650ns, 750ns, and 850ns; and a screenshot of the waveform.

12. Open Schematic (under "RTL Analysis"->" Elaborated Design") and analyze the output. Include the schematic in the report.

13. Create a new module to implement a 32-bit counter (Figure 7.) with module name, inputs and outputs as shown in the Figure 8. The first step is resetting the counter by supplying 1 to the "reset" input: irrespective of value of clk or count_en, the value of the register Q will be all 0s after "reset" becomes 1. Also, if "reset" is 0, only when "count_en" is 1, the counter will count: on every positive edge of "clk" signal the current value of the register will be incremented (1 will be added to the current vlaue). For example, after the rest the counter will have values 0, 1, 2, 3… that will be updated on every positive edge of the clock. Moreover, if "reset" is 0, when "count_en" is 0, the current content of the counter will be preserved even on positive edge of "clk" signal. Please refer to accompanying video tutorials on BB for more details.

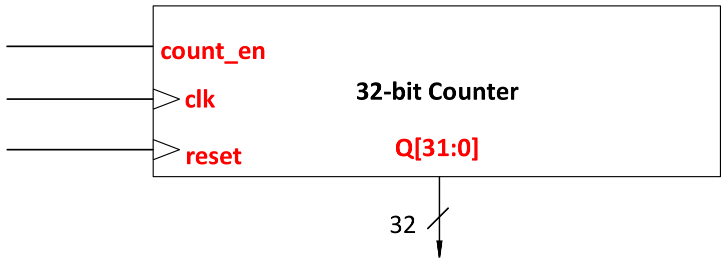
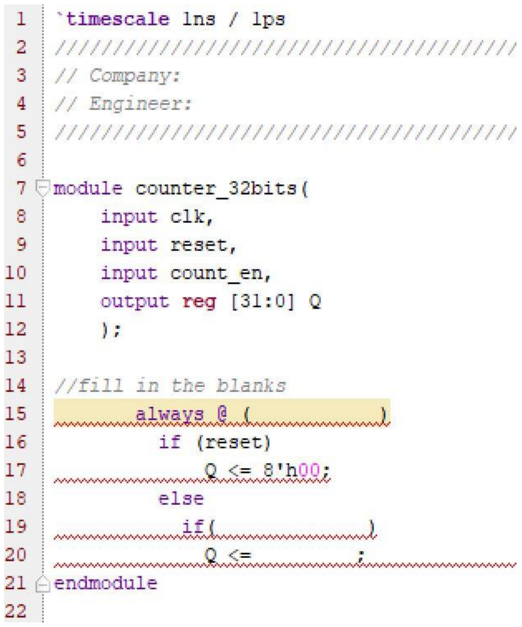


Figure 7. Top level view of a 32-bit counter

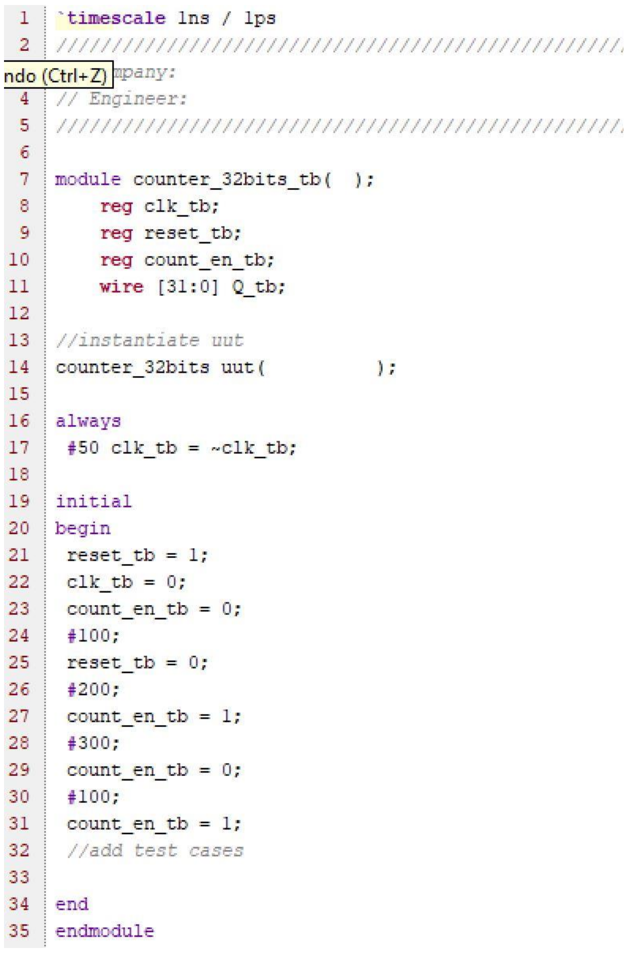
14. In the class, we drew detail schematics of different multi-functional registers that contain a basic register and a multiplexor. When coding in Verilog, you **need not** to make an instance of a mux and an instance of a basic register. Verilog allows you to implement the same functionality using an **if** statement in the code that corresponds to update of the register: please see lines 19 and 20 in the Figure 8. To implement the counter, you only need to add 1 to the current value of Q in the line 20 of the code.

15. Create a testbench for the 32-bit counter as per Figure 9. Instantiate the uut using the variables defined at the beginning of the testbench. Add more tests inside the "initial" statement:
    - after 300ns: initialize count_en_tb to 0
    - after another 150ns: initialize count_en_tb to 1
    - after another 75ns: initialize reset_tb to 1
    - after another 50: initialize reset_tb to 0



```
`timescale 1ns / 1ps
////////////////////////////////////////
// Company:
// Engineer:
////////////////////////////////////////

module counter_32bits(
    input clk,
    input reset,
    input count_en,
    output reg [31:0] Q
    );

//fill in the blanks
       always @ (              )
         if (reset)
             Q <= 8'h00;
         else
           if(                )
             Q <=          ;
endmodule

```

Figure 8. Module counter_32bits, its inputs and outputs, and the module body.



```
`timescale 1ns / 1ps
////////////////////////////////////////////////////.
mpany:
// Engineer:
////////////////////////////////////////////////////.

module counter_32bits_tb(  );
    reg clk_tb;
    reg reset_tb;
    reg count_en_tb;
    wire [31:0] Q_tb;

//instantiate uut
counter_32bits uut(          );

always
 #50 clk_tb = ~clk_tb;

initial
begin
 reset_tb = 1;
 clk_tb = 0;
 count_en_tb = 0;
 #100;
 reset_tb = 0;
 #200;
 count_en_tb = 1;
 #300;
 count_en_tb = 0;
 #100;
 count_en_tb = 1;
 //add test cases

end
endmodule
```

Figure 9. Testbench counter_32bits _tb

16. Simulate the testbench for enough time to show all the new test cases and observe the outputs in the simulation window. Include in the report: a table with an expected value between 1,000ns and 1,400ns at 50ns intervals; and a screenshot of the waveform.

17. Open Schematic (under "RTL Analysis"->" Elaborated Design") and analyze the output. Include the schematic in the report.

18. Next step is to connect the 32-bit counter to the FPGA board. The counter has a 32-bit output, but the board has only 16 distinct LEDs. Therefore, you need to select either the lower 16-bits, or the upper 16-bits to be displayed, using a 16-bit 2-to-1 mux, as shown in the Figure 10. Then, you will connect the output of the mux to the LEDs and choose the lower or the upper 16-bits to display using the select signal of the mux.

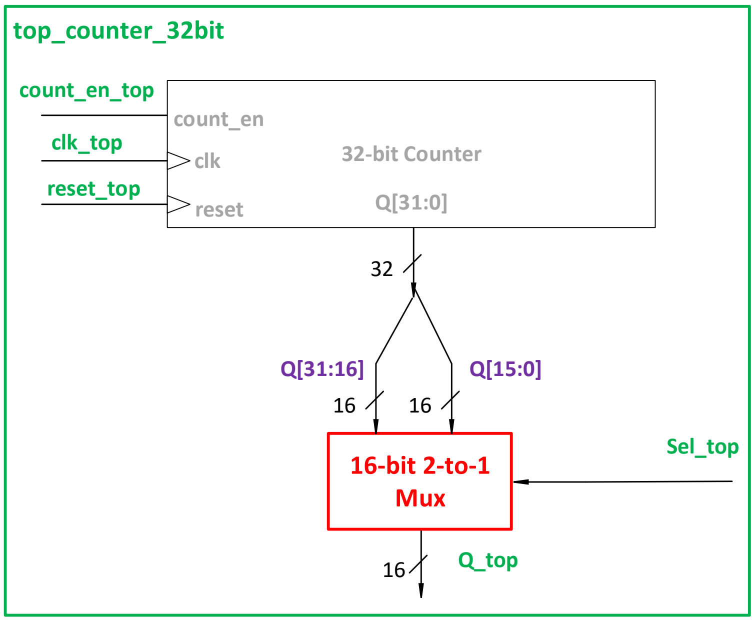
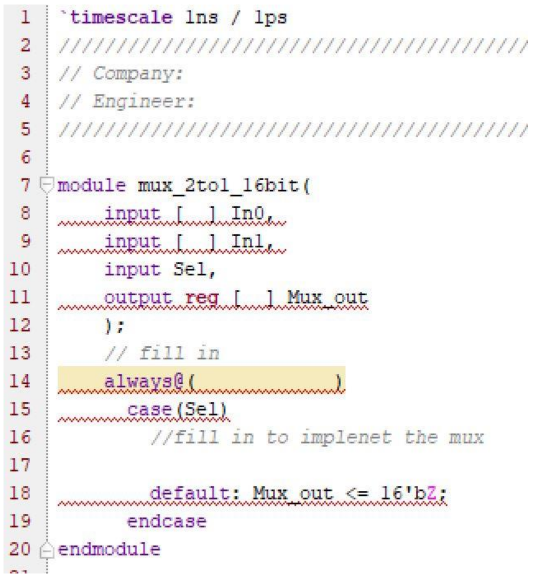
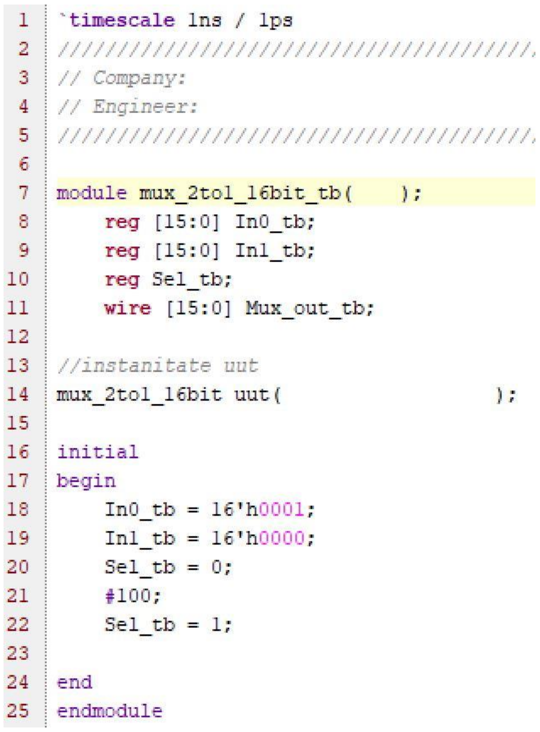


Figure 10. Top level design top_count_32bit that allows us connection of the 32-bit counter to 16 LEDs

19. First, you need to create a 16-bit 2-to-1 mux and test it, before incorporating it into the top_counter_32_bit. Create appropriate modules as per Figure 11 a) and b), while keeping all the names as given.



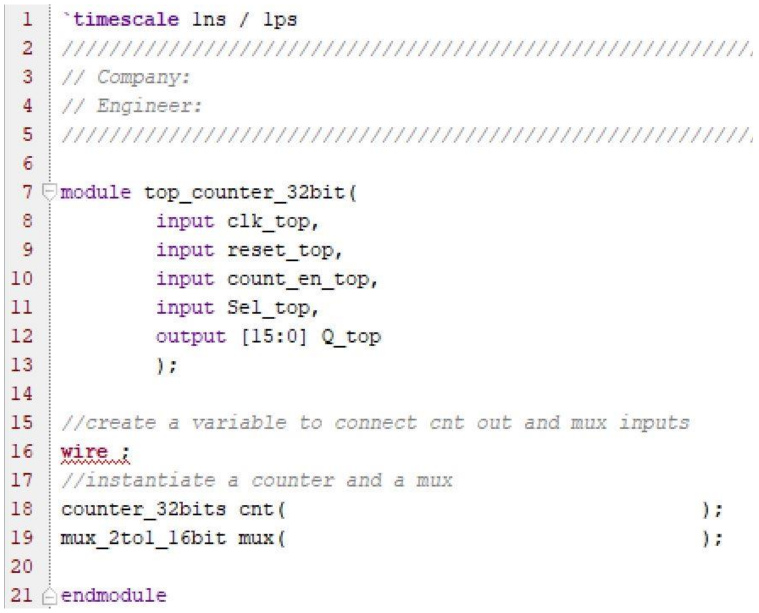
```
`timescale 1ns / 1ps
////////////////////////////////////////
// Company:
// Engineer:
////////////////////////////////////////

module mux_2to1_16bit(
    input [  ] In0,
    input [  ] In1,
    input Sel,
    output reg [  ] Mux_out
    );
    // fill in
    always@(              )
      case(Sel)
        //fill in to implenet the mux

        default: Mux_out <= 16'bZ;
      endcase
endmodule
```



```
`timescale 1ns / 1ps
////////////////////////////////////////
// Company:
// Engineer:
////////////////////////////////////////

module mux_2to1_16bit_tb(    );
    reg [15:0] In0_tb;
    reg [15:0] In1_tb;
    reg Sel_tb;
    wire [15:0] Mux_out_tb;

//instanitate uut
mux_2to1_16bit uut(                    );

initial
begin
    In0_tb = 16'h0001;
    In1_tb = 16'h0000;
    Sel_tb = 0;
    #100;
    Sel_tb = 1;

end
endmodule
```

Figure 11 a) 16-bit 2-to-1 mux module; b) corresponding tesbench

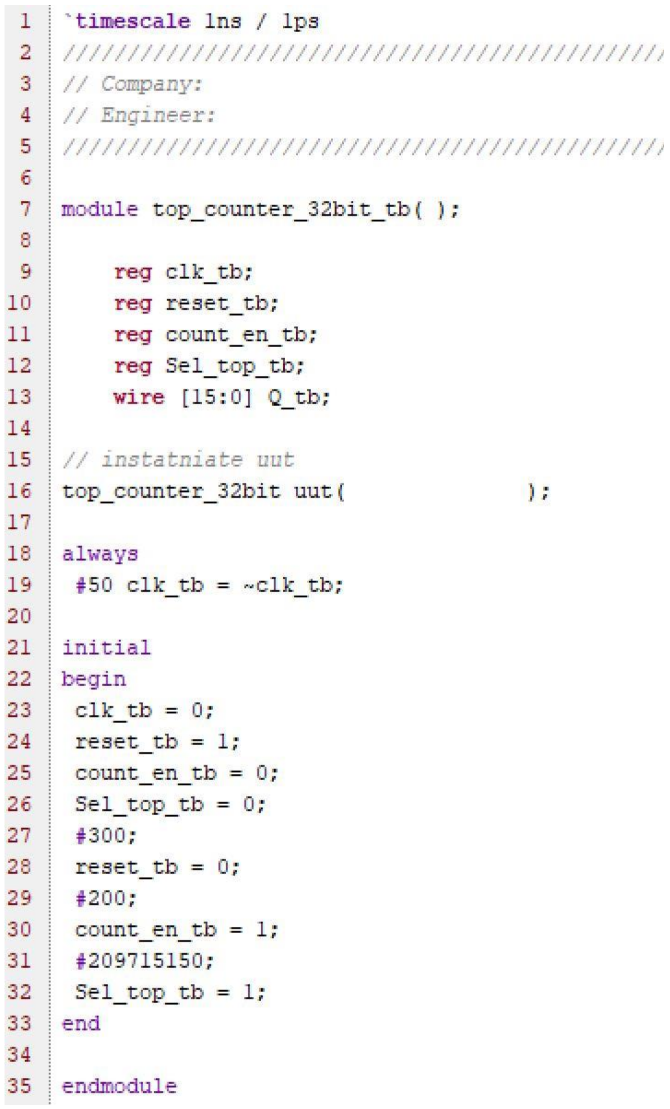
20. Create a new module top_counter_32bit (Figure 10.) with module name, inputs and outputs as shown in the Figure 12. Fill in the blanks to instantiate one instance of a counter_32bits and an instance of a 16-bit 2-to-1 mux, while connecting them to implement the schematic given in the Figure 10.



```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////
// Company:
// Engineer:
//////////////////////////////////////////////////////////

module top_counter_32bit(
        input clk_top,
        input reset_top,
        input count_en_top,
        input Sel_top,
        output [15:0] Q_top
        );

//create a variable to connect cnt out and mux inputs
wire ;
//instantiate a counter and a mux
counter_32bits cnt(                                   );
mux_2to1_16bit mux(                                   );

endmodule
```

Figure 12. Module skeleton for top_counter_32bit

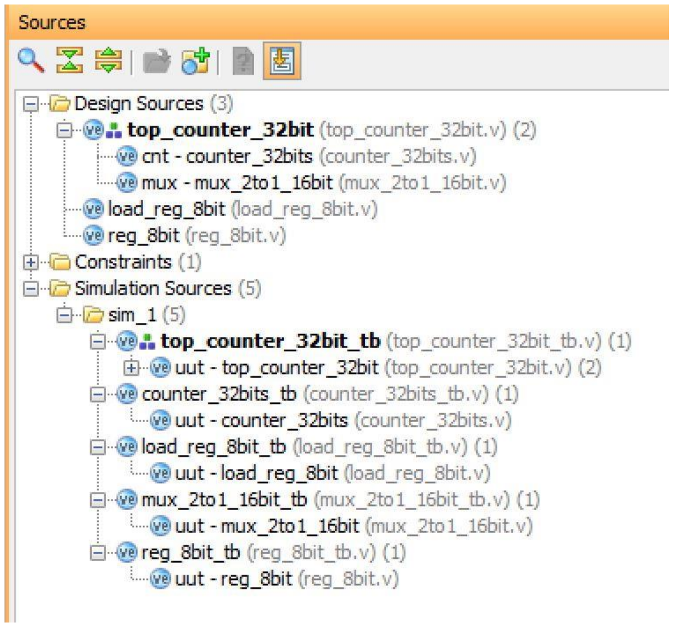
21. Create a testbench for the top_counter_32bit as per Figure 13. Instantiate the top_counter_32bit using the variables defined at the beginning of the testbench. Study the code inside "initial" statement, and decide the minimal time needed for the simulation to be able to see all changes.



```
`timescale 1ns / 1ps
//////////////////////////////////////////////////
// Company:
// Engineer:
//////////////////////////////////////////////////

module top_counter_32bit_tb( );

    reg clk_tb;
    reg reset_tb;
    reg count_en_tb;
    reg Sel_top_tb;
    wire [15:0] Q_tb;

// instatniate uut
top_counter_32bit uut(               );

always
 #50 clk_tb = ~clk_tb;

initial
begin
 clk_tb = 0;
 reset_tb = 1;
 count_en_tb = 0;
 Sel_top_tb = 0;
 #300;
 reset_tb = 0;
 #200;
 count_en_tb = 1;
 #209715150;
 Sel_top_tb = 1;
end

endmodule
```

Figure 13. Testbench for top_counter_32bit

22. When all modules created and added to the project, the "Sources" window will show the hierarchy as in Figure 14.


Sources

Design Sources (3)
top_counter_32bit (top_counter_32bit.v) (2)
cnt - counter_32bits (counter_32bits.v)
mux - mux_2to1_16bit (mux_2to1_16bit.v)
load_reg_8bit (load_reg_8bit.v)
reg_8bit (reg_8bit.v)
Constraints (1)
Simulation Sources (5)
sim_1 (5)
top_counter_32bit_tb (top_counter_32bit_tb.v) (1)
uut - top_counter_32bit (top_counter_32bit.v) (2)
counter_32bits_tb (counter_32bits_tb.v) (1)
uut - counter_32bits (counter_32bits.v)
load_reg_8bit_tb (load_reg_8bit_tb.v) (1)
uut - load_reg_8bit (load_reg_8bit.v)
mux_2to1_16bit_tb (mux_2to1_16bit_tb.v) (1)
uut - mux_2to1_16bit (mux_2to1_16bit.v)
reg_8bit_tb (reg_8bit_tb.v) (1)
uut - reg_8bit (reg_8bit.v)

Figure 14. Sources window show correct hierarchy

23. Simulate the testbench and observe the outputs in the simulation window (run the simulation for additional time to observe all test cases). What is being shown after the following lines of code:
    #209715150;
     Sel_top_tb = 1;
    Include your answer in the report.

24. Open the schematic and include the screenshot of the schematic "as-is" in the report. Expand both module instances ("cnt" and "mux") by clicking the + sign as shown on the Figure 15. Include the screenshot of the new schematic in the report.

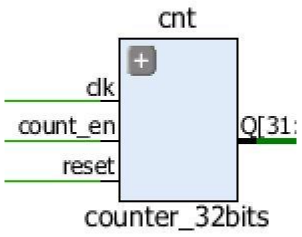


Figure 15. Expanding the instance of the module

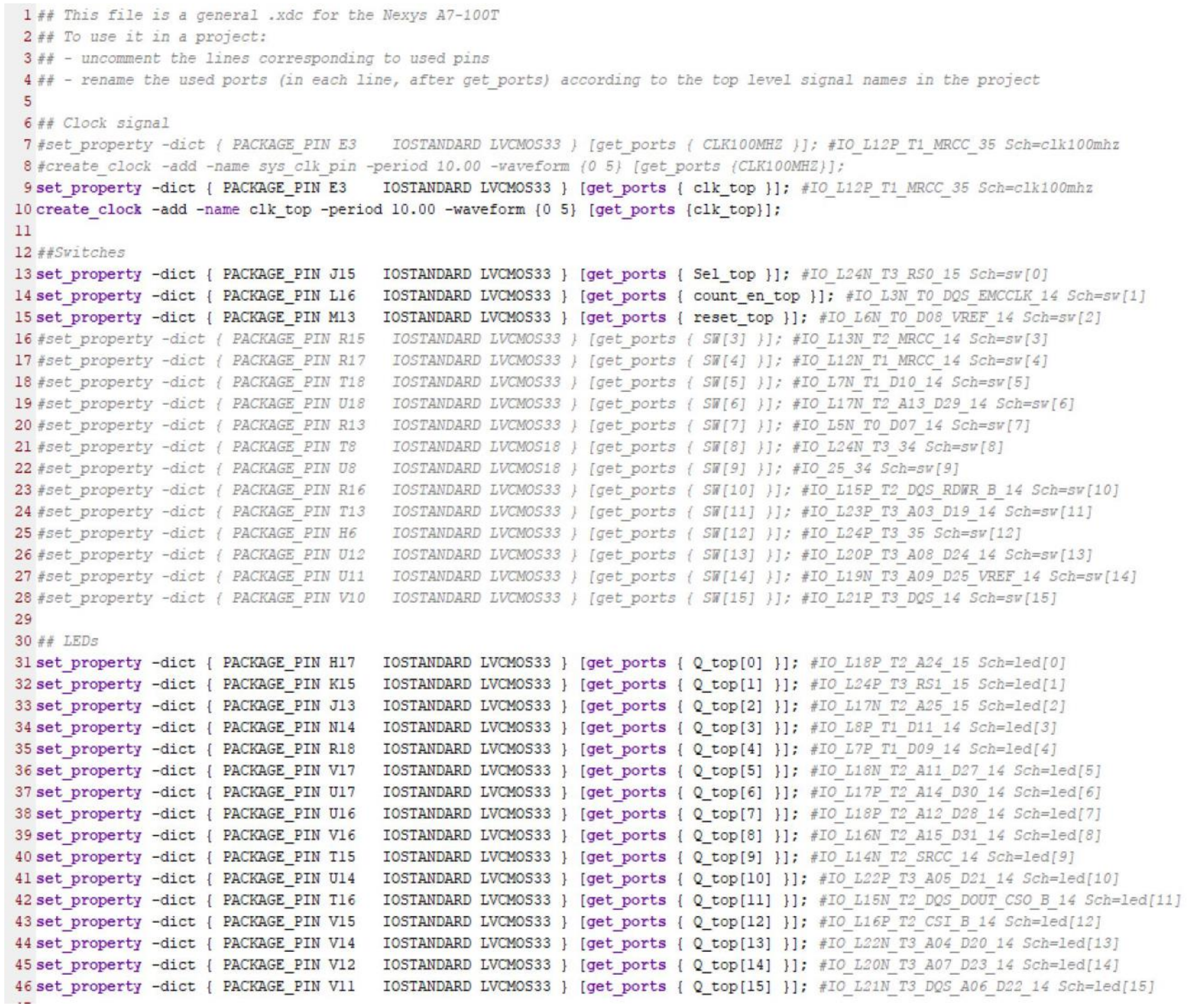
Synthesis:

25. Make sure top_counter_32bit and top_counter_32bit_tb are both "set as top", and that top_counter_32bit is functionally correct.

26. Add a constraint file using "Add source" option "Add or create constraints", press "Next". Name the file "top_counter_32bit_NexysA7-100T", press "OK" and "Finish". Download

file “MasterConstraint_NexysA7-100T.txt” from Lab 4 folder on BB and save it locally. Open the file with the text editor (Notepad or WordPad), copy its content to clipboard, and paste the content inside the “top_counter_32bit_NexysA7-100T.xdc” file in the editor.

27. Modify the file to map the inputs of the top_counter_32bit to the switches, and the outputs to the LEDs, by removing # at the beginning of an appropriate line, and changing parameters for get_ports. Make sure to **map the clock** as well, as shown in the Figure 16. at lines 9 and 10. Save the file.



```
 1 ## This file is a general .xdc for the Nexys A7-100T
 2 ## To use it in a project:
 3 ## - uncomment the lines corresponding to used pins
 4 ## - rename the used ports (in each line, after get_ports) according to the top level signal names in the project
 5
 6 ## Clock signal
 7 #set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { CLK100MHZ }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
 8 #create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK100MHZ}];
 9 set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clk_top }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
10 create_clock -add -name clk_top -period 10.00 -waveform {0 5} [get_ports {clk_top}];
11
12 ##Switches
13 set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { Sel_top }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
14 set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { count_en_top }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
15 set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { reset_top }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
16 #set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { SW[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
17 #set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { SW[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
18 #set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { SW[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
19 #set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { SW[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
20 #set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { SW[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
21 #set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports { SW[8] }]; #IO_L24N_T3_34 Sch=sw[8]
22 #set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports { SW[9] }]; #IO_25_34 Sch=sw[9]
23 #set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports { SW[10] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
24 #set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports { SW[11] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
25 #set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports { SW[12] }]; #IO_L24P_T3_35 Sch=sw[12]
26 #set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports { SW[13] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
27 #set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports { SW[14] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
28 #set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports { SW[15] }]; #IO_L21P_T3_DQS_14 Sch=sw[15]
29
30 ## LEDs
31 set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { Q_top[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
32 set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { Q_top[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
33 set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { Q_top[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
34 set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { Q_top[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
35 set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { Q_top[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
36 set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { Q_top[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
37 set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { Q_top[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
38 set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { Q_top[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
39 set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { Q_top[8] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
40 set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports { Q_top[9] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
41 set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports { Q_top[10] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
42 set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { Q_top[11] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
43 set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { Q_top[12] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
44 set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports { Q_top[13] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
45 set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports { Q_top[14] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
46 set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports { Q_top[15] }]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]
```

Figure 16. XDC file

28. Follow the procedure outlined in the previous labs: “Run Synthesis”, “Run Implementation”, “Generate Bitstream”, connect and power on the Nexys A7-100T board, and finally “Open Hardware Manager”. “Open Target” and chose “Auto connect” and “Program device”.

29. The “light show” on the board will stop. Move the three rightmost switches (see the .xdc file describing the connections) to test the design: reset the counter, enable counting and change from selecting lower to selecting upper 16-bits of the counter. From time to time disable counting. What do you see when lower 16-bit are selected? How about when you disable counting while the lower 16-bit are selected? Repeat the procedure for the upper 16-bit and write your observation in the report

30. Close “Hardware Manager” by pressing X on the blue nav bar in top right (do not need to close the project or Vivado). Move “POWER” switch on the board to “OFF” position and unplug the board.

What to submit:

Upload to Lab_7 Dropdox the following files:

a. reg_8bit.v , reg_8bit _tb.v,
b. load_reg_8bit.v , load_reg_8bit.v,
c. mux_2to1_16bit.v , mux_2to1_16bit_tb.v,
d. counter_32bits.v , counter_32bits_tb.v,
e. top_counter_32bit.v, top_counter_32bit_tb.v, top_counter_32bit_NexysA7-100T.xdc, and
f. your lab report.

Refer to the previous labs to locate the sources within the project.

Questions:

The questions to be asked during the demo will be similar, but not limited to, the questions listed below:

1. Show the design being simulated: explain the waveform seen during the simulation.

2. Observe all testbenches: explain the inputs they generate?

3. Explain how the implemented designs work on FPGA board. Is this something you expected? Why or why not?

4. Explain sensitivity list for each register for **always** statement on the example of encoder and decoder.

5. Why are you able to see upper 16 bits changing the value and why are you not able to see the lower 16 bits change the values?

6. What happens when you disable counting while showing lower/upper 16 bits? Why? Is this something that you expected?