

Lab 5 Tutorial - Introduction to Counters and CPLDs

Before we begin the tutorial for the digital counter lab, take a moment to ensure that you are familiar with the following concepts:

1. Utilizing a finite state-machine to design a circuit.
2. Implementing and simulating a design utilizing ISE and ModelSim.
3. Configuring the CPLD using the Xilinx iMPACT device programmer.

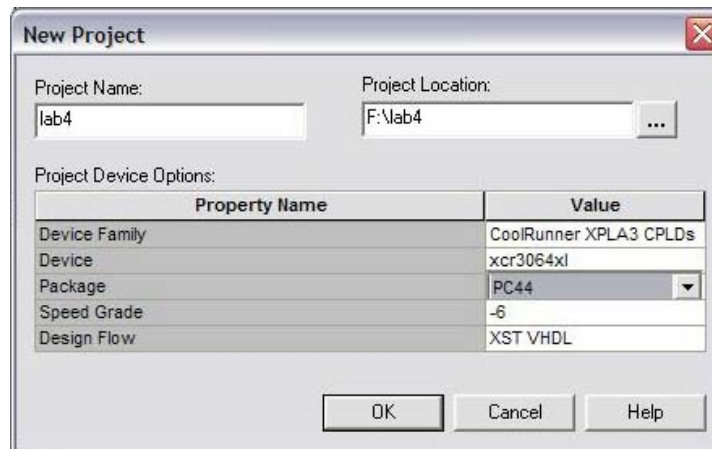
In addition to these basic concepts, this lab will introduce you to several new concepts, including the following:

1. Creating a project that utilizes multiple VHDL files.
2. Routing signals in and out of the device to facilitate I/O.
3. Designing and implementing several different types of counters.

As always, this tutorial will be divided into easy-to-follow steps. Make sure to print the Laboratory Worksheet and to fill it out as you progress in the Tutorial.

Part 1: Create a New Project

By now this should be familiar to you. Create a new project in Project Navigator and make sure the settings are exactly like the ones shown below:



Part 2: Creating Independent VHDL Modules

Once you have created the project, download this VHDL file , save it as XCR7SEG.VHD and add it to your project as a VHDL Module. This file drives the two seven segment displays that you see on the XCR board. Read the code and identify the process described on it. Make sure to answer the questions about it in the laboratory worksheet. In Part 3,

we will show you how to incorporate this separate VHDL file into your top-level design. For now, set the XCR7SEG file aside and create another new VHDL file. You will need to make sure the following standard declarations are at the top:

```
-----
-- XCR_COUNT.VHD -- Lab 5 Tutorial
-----
-- Author: <Your name goes here>
-----
-- This is the VHDL file for the Lab 4
-- tutorial counter. The VHDL you will write for the lab
-- itself should be similarly commented. This tells you
-- at a glance what file you're dealing with.
--
-- Inputs: <Your inputs go here.>
-- Outputs: <Your outputs go here>
--
-----
-- Revision History:
-- <Date>( <Your name> ): Created.
-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

Now we need to come up with a suitable entity statement:

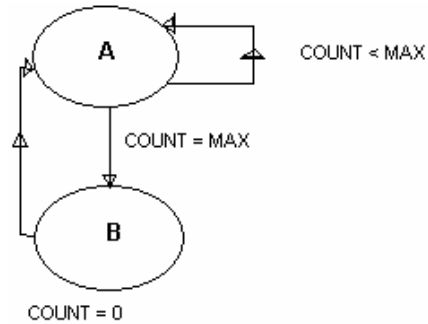
```
entity XCR_COUNT is
PORT
(
    sclk_h : in std_logic; -- System clock.
    sw0_h : in std_logic; -- Switch to control the counter.
    sw1_h : in std_logic; -- Another switch to control the counter.

    data_out_h : out std_logic_vector(3 downto 0)
);
end XCR_COUNT;
```

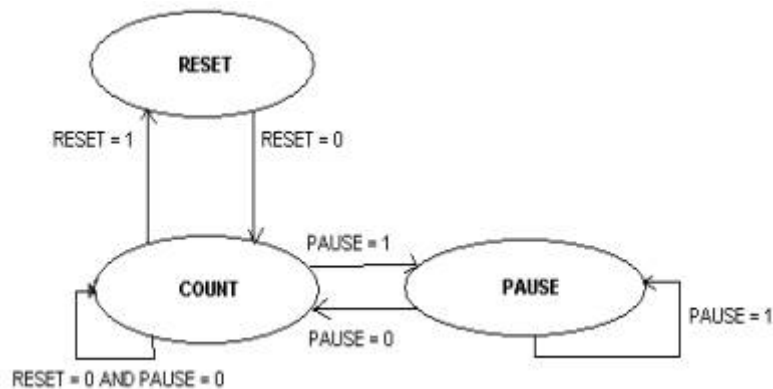
Now, before we continue, go back to the comment block at the top and fill in all the information asked for in the < ... > brackets.

Designing the Counter

Before we can develop a counter circuit, we must briefly touch upon how a counter is supposed to function. As you may recall in the lab lecture, a counter is a registered circuit. That is, it has some logic that allows it to store information. This also implies that the counter circuit is a finite state machine. Therefore, it has several discrete states that it can be in. These states can only change at the rising or the falling-edge of the system clock. For a simple counter, we may have only two states, a state A, and a state B. In state A, the counter will continue to increment as long as the value of the count is less than the maximum value. When the counter reaches its maximum value, it goes into state B, where it is reset to zero and returns to state A.



The counter we will be showing you in the tutorial will have three separate states. Take a moment to study the image below:



Now, how might we implement this? Recall that in VHDL there are two potential ways to implement a FSM. One way is to use case-statements, where each case represents a state. This was the approach we took in the sequence detector lab. Another way of implementing a FSM is to use if/else statements. This is the approach we will take in this lab. Now we will show you how this is done. Copy and paste the following VHDL code into your file now:

```

architecture Behavioral of XCR_COUNT is
-----
-- Component Declarations
-----

-----
-- Signal Declarations
-----

signal qtemp: std_logic_vector(3 downto 0);
--temporary variable for output q[3..0]

signal clkdiv: std_logic_vector(2 downto 0);
--Clock divider, adjust to suit your needs.

```

```

signal dclk: std_logic;
signal sclk: std_logic;

--type state_type is (A, B, C, D);
--signal state, next_state : state_type;

begin
    CLK_DIV : process (sclk_h)
    begin
        if sclk_h = '1' and sclk_h'Event then
            clkdiv <= clkdiv + 1;
        end if;
    end process;

    -- Decade counter driven by the above clock divider to generate
    -- the digits to go out to the seven segment displays.

    dclk <= clkdiv(2);

```

Here we have a clock divider. This enables the counter to count slower than the LED display is refreshing itself. If we ran both the counter and the display at the same speed, the count would pass by much too quickly for you to see it. Below, you will find the actual guts of our counter:

```

COUNT_PROC : process(sclk_h,sw0_h,sw1_h) -- Process definition
begin
    if sw0_h='1' and sw1_h = '0' then
        qtemp<="0000"; -- Asynchronous reset.
    elsif sw0_h = '0' and sw1_h = '1' then
        qtemp<=qtemp+0; -- Asynchronous count pause.
    else
        -- if sclk'event and sclk='1' then -- Counting state for simulation.

        if dclk'event and dclk='1' then
            -- Counting state for implementation

```

Before we give you the rest of the counter, take careful note of the if-statement here. Note that we have one of them commented out. We've commented out the if-statement that controls the counter for the ModelSim simulator. If we did not do this, the counter will be running at the same speed as the system clock, and would count at a speed that is almost too fast to observe. However, for ModelSim, we will want to run the counter at the same speed as the system clock. Swap the comment so the if-statement containing **sclk'event and sclk='1'** is uncommented. You will need to change the comments around again before you actually implement this design on the CPLD. Now, enter in the rest of the VHDL as shown below:

```

        if qtemp<15 then
            qtemp<=qtemp+1; -- Counter increase
        else
            qtemp<="0000"; -- Return the zero state
        end if;
    end if;

    data_out_h<=qtemp; -- Output
end if;

```

```
end process; -- End Process
end Behavioral; -- End module.
```

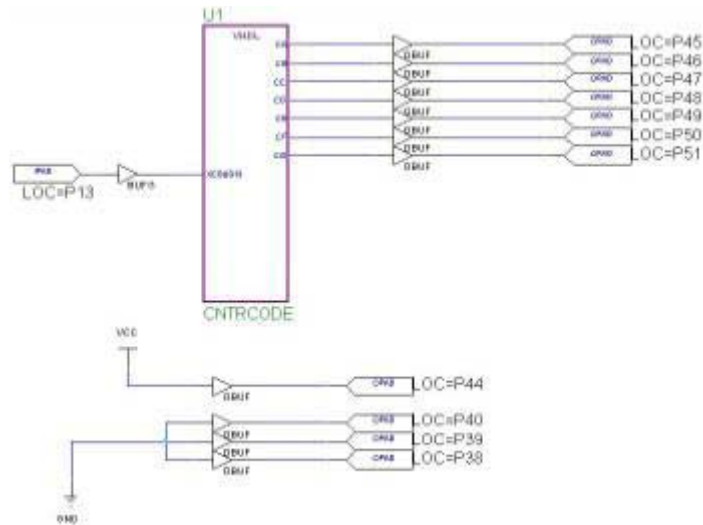
Carefully examine our if/else statements. Can you identify the three FSM states here? Now save this file as XCR_COUNT.VHD and add it to your project as a VHDL Module. This file now contains the information needed to implement a counting module. And we have already created a VHDL file that implements a display module. However, we do not have something we can program onto the CPLD just yet. That will come in Part 3.

Part 3: Multiple-File Designs in VHDL

In this section, we will discuss how to create a VHDL design with multiple components. However, before we can do this, we should discuss the basic history of digital logic design.

Early in the days of digital logic design, everything was designed using schematics. These schematics indicated what logic we were using, and how it was looked up. Other schematics were then used to indicate how the individual devices were to be laid out.

A Sample Schematic:



In fact, schematics are still in use today. Logic design suites such as the one furnished by Mentor Graphics have schematic-capture support. In earlier labs, we have asked you to develop schematics before you implemented your designs in 7400 logic.

The current trend in the industry is VHDL and not schematic capture. This is due to the ease of scalability. Large scale schematic capture projects are just not reasonable. It is possible to create HDL designs that incorporate more than one module. For example, if we were designing a computer system, it might require the following modules:

1. A CPU module, which will have an instruction register, an ALU, and all the associated data and control lines.
2. An I/O module that interfaces the CPU to the outside world.
3. A memory module that stores the results of what the CPU is doing.

All of these would be incorporated into a top-level VHDL design. This is exactly what we will be doing with our counter. We will take the counting module, and the display module, and we will incorporate them into a single top-level VHDL file. This top-level VHDL file replaces the schematic found in earlier digital logic designs.

Creating the Top-Level Design

When designing in VHDL, ideally, the top-level design should be as simple as possible. All it should do is route signals between the modules that are incorporated into the project. In this regard, it is similar to a well-designed C/C++ program, where all the work takes place inside functions outside of `main()`, which is the top-level part of the program. Create a new VHDL file and add the following lines to it:

```
-----
-- XCR_TOP.VHD -- Lab 5 Tutorial
-----
-- Author: <Your name goes here>
-----
-- This is the top-level VHDL file for the Lab 4
-- tutorial. The VHDL you will write for the lab
-- itself should be similarly commented. This tells you
-- at a glance what file you're dealing with.
--
-- Inputs: <Your inputs go here.>
-- Outputs: <Your outputs go here>
--
-----
-- Revision History:
-- <Date>( <Your name> ): Created.
-----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity TOP is
PORT
(
    sys_clk : in std_logic; -- System clock.
    count_clk : in std_logic; -- Counter clock.
    in_sw0_h : in std_logic; -- Input switch 1.
    in_sw1_h : in std_logic; -- Input switch 2.

    anodes_out : out std_logic_vector(1 downto 0); -- Anode enables.
    disp_out : out std_logic_vector(6 downto 0) -- Display out.
);
end TOP;
```

Note here that our top level design has all the inputs necessary to drive the counter module, and the outputs necessary to drive the display using the display module. Now, we need to get these modules into the top-level file. Add the following VHDL to your file now:

```
architecture Behavioral of TOP is
-----
-- Component Declarations
-----

component xcr7seg is
Port (
    mclk : in std_logic;
    dig_l_H : in std_logic_vector(3 downto 0);
    dig_u_H : in std_logic_vector(3 downto 0);

    anode : out std_logic_vector(1 downto 0);
    ssg_out_H : out std_logic_vector(6 downto 0)
);
end component;

component XCR_COUNT is
PORT
(
    sclk_h : in std_logic; -- System clock.
    sw0_h : in std_logic; -- Switch to control the counter.
    sw1_h : in std_logic; -- Another switch to control the counter.

    data_out_h : out std_logic_vector(3 downto 0)
    -- For ModelSim simulator.
);
end component;
```

If you inspect these component declarations closely, you will notice that we have simply taken the entity declarations from the our previous VHDL files, XCR_COUNT.VHD and XCR7SEG.VHD and dropped them in as components of this top-level file. All the component declarations do is tell the top-level design that we will be using these components at some point in our design. That is what you will do when you design your own counter. Now, we need the internal signals that will connect our modules together and to the outside world.

```
-----
-- Signal Declarations
-----

signal intclk : std_logic; -- Clock signal driving counter.
signal dispclk : std_logic; -- Clock signal driving display.
signal reset : std_logic; -- Input signal driving reset.
signal freeze : std_logic; -- Input signal driving counter pause.
signal data_bus : std_logic_vector(7 downto 0); -- Self-explanatory.
signal output_bus1_h : std_logic_vector(6 downto 0);
signal anode_output : std_logic_vector(1 downto 0);
```

In a minute, we will use these signals to connect our included modules together, but first, we have to actually use the included modules in our design. That is, we need to declare instances of each module. This is done in the code below:

```

begin
-----
-- Component instantiations
-----

C_7SEG : xcr7seg
port map (

    mclk => dispclk,
    dig_l_H => data_bus(7 downto 4),
    dig_u_H => data_bus(3 downto 0),
    anode => anode_output,
    ssg_out_H => output_bus1_h
);

C_COUNT : XCR_COUNT
port map (

    sclk_h => intclk,
    sw0_h => reset,
    sw1_h => freeze,
    data_out_h => data_bus(3 downto 0)
);

```

Note how we declared a single instance of each component. We took the inputs and outputs of both components, and tied them to the internal signals we declared earlier. Note that now the components C_7SEG and C_COUNT are tied together on the same internal bus, called *data_bus*. Finally, we need to connect our final design to the outside world:

```

-----
-- System routing.
-----

dispclk <= sys_clk; -- Both the display and internal clocks
intclk <= sys_clk; -- are tied to the system clock.
reset <= in_sw0_h; -- Reset gets the first switch.
freeze <= in_sw1_h; -- Our pause gets the second switch.
anodes_out <= anode_output;
-- Drives the anodes for the 7 segment displays.
disp_out <= output_bus1_h;
-- Drives the output to the 7 segment displays.

end Behavioral;

```

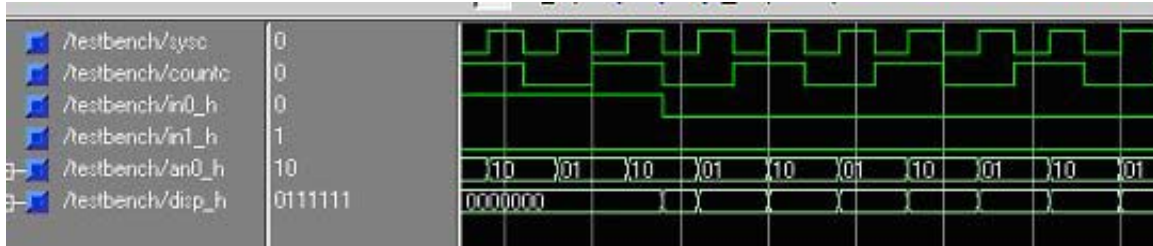
Save this file as **XCR_TOP.VHD** and import it into your project as a VHDL Module. As this file is your top-level design, this will be the file that you synthesize and implement!!!!!! At this point, check the file for syntax errors. Once you have fixed any syntax errors, it will be time to simulate the project in ModelSim.

Part 4: Simulating the Project

Recall that in the previous labs, if you wanted to simulate something in ModelSim, you had to write a VHDL test bench. In that regard, this project is no different from the other

projects. Except in this instance, we will show you how to test both the top-level design, and one of the components which makes up the top level design. For now, download the entire test bench here. Now save it as **TOP_TB.VHD** and import it into your project as a VHDL Test bench. Now to simulate the top-level design, merely highlight the test bench file, and double-click on Simulate Behavioral VHDL Mode.

If you are successful, you should have the waveform screen available. Maximize it and tell ModelSim to Zoom Full. Then, zoom in several times until you get something that looks like the following:



At this point, print out your ModelSim waveform so you can attach them to your final report. At the end of the tutorial, you will be given an opportunity to modify the VHDL test bench so you can simulate the counter itself. However, we need to prepare our design to be downloaded onto the device.

Before we can do that, however, remember some time back when we changed the following two lines in the XCR_COUNT.VHD file from this:

```
-- if sclk'event and sclk='1' then -- Counting state for simulation.
if dclk'event and dclk='1' then -- Counting state for implementation
```

To this:

```
if sclk'event and sclk='1' then -- Counting state for simulation.
-- if dclk'event and dclk='1' then -- Counting state for implementation
```

Before you can map your design out to the device, you need to change those two lines back to their original state.

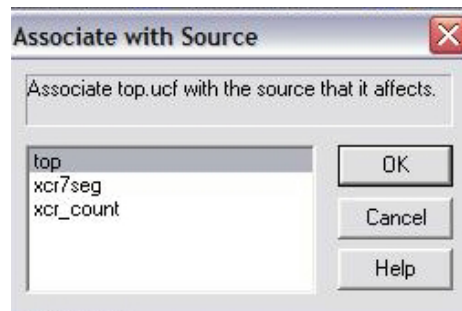
Part 5: Mapping Your Design into the Device

Once you have successfully simulated the design in ModelSim, it is time to map it to your target device. To do this, make sure you have your top-level VHDL file highlighted in the Sources In Project window. Next, expand the Design Entry Utilities portion of the Processes for Current Source window. Once there, you should see something called User Constraints. Expand that to reveal three potential ways of mapping your design out to the target device.

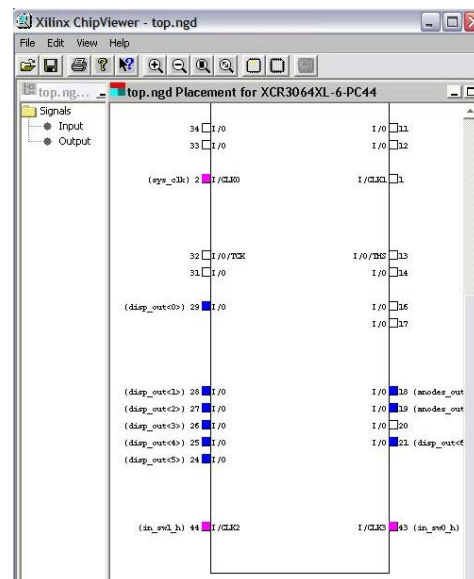
In Lab 3, we showed you how to assign pins using the Constraints Editor. Here, we are going to show you a more basic way of doing it. That is, you will actually edit the UCF file itself. Create a new file and paste the following lines into it.

```
NET anodes_out<0> LOC=P18;
NET anodes_out<1> LOC=P19;
NET disp_out<6> LOC=P21;
NET disp_out<5> LOC=P24;
NET disp_out<4> LOC=P25;
NET disp_out<3> LOC=P26;
NET disp_out<2> LOC=P27;
NET disp_out<1> LOC=P28;
NET disp_out<0> LOC=P29;
NET in_sw0_h LOC=P43;
NET in_sw1_h LOC=P44;
NET sys_clk LOC=P2;
```

Save the file as “**top.ucf**”. Then go to **Project, Add Source**. The following dialog box will appear.



Choose top and click OK. Now, we want to look at what the pin assignments as they would look on the chip. To do this we want to open Chipviewer. In the Processes for Current Source expand User Constraints. Double click Assign Package Pins. This will bring up the following window.



Even though we did not do it here, you could have used Chipviewer to assign your pins. If you had not yet assign your pins when you opened Chipviewer you would find all of

the variables on the right under Signals. You could then drag and drop each variable onto the pin you wanted to assign to it. You are now ready to implement the design.

Part 6: Implementing Your Design

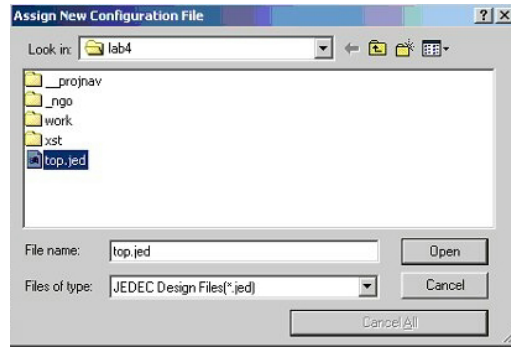
At this point, we are ready to implement our design. To do this, simply double-click on the **Generate Programming File** option in the **Processes for Current Source** window. This will run, (or re-run) all the synthesis steps, all the implementation steps, and finally, it will generate a file that will be used to program the actual device. If everything is successful, the **Synthesize** option, the **Implement Design** option, and the **Generate Programming File** option will show either green checkmarks, or yellow exclamation points. The yellow exclamation points indicate that there were non-fatal warnings that have come up while ISE was processing your design. Usually, they can be ignored, but if a design is not functioning the way you would expect it to, or the way it worked in simulation, then you will want to take a closer look at the warnings. If you have red Xs at any point, these signify fatal errors. If you get those, you will have to correct your design before you can try to implement it again.

Part 7: Programming the Device

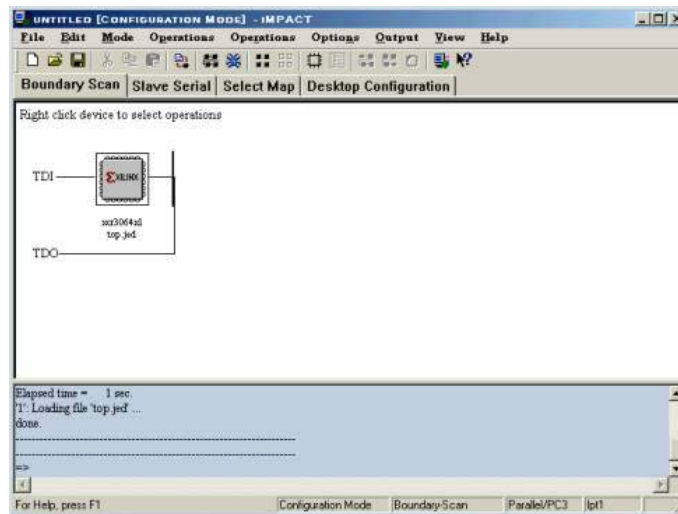
Once you have successfully implemented your design, expand the **Generate Programming File** option in the **Processes for Current Source** window. This will give you the option to configure the device using **iMPACT**. Before you launch **iMPACT** however, make sure that the XCR board is turned on, and that you have made sure that the switchbox on top of your computer is set to CPLD. Once you have done this, go ahead double click Configure Device (**iMPACT**).

An Operation Mode Selection dialog box will come up. Choose Configure Devices and click **Next**. Then a Configure Devices dialog box will pop up. Choose Boundary-Scan Mode and click **Next**. Then a Boundary-Scan Mode Selection dialog box will come up. Choose **Automatically connect to cable and identify Boundary-Scan chain** and click **Finish**. Click **Ok**.

A new window like below will pop up. Browse to the directory of your project and open **top.jed**.



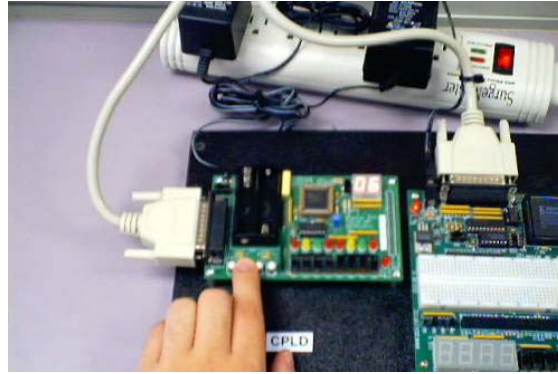
Move your mouse cursor over the chip icon and right-click once. You should be able to select Program in the menu that appears. Select it, by clicking on it once. Accept all the default options in the **Program Options** dialogue box and click Ok.



Immediately your device should go blank. After a few seconds, you should see the following:



The left digit will remain zero the whole time, while the right digit will count from 0xh to Fxh (which is 0 to 15 in decimal.) The display will flicker slightly. If you press and hold down the second button from the left, as shown below:



You should notice that the counter stops counting and remains stopped as long as you hold down that button. Once you release the button, the counter will pick up where it left off. On the other hand, if you press the second button from the right as shown below:



You should notice that things are different when you release the button. This time, the counter starts counting again from zero. That button is the reset button. At this point, you have completed all the steps in the tutorial.

Checkpoint

Make sure you have the following items ready to turn in with your report:

1. Copies of your VHDL top-level file, and your VHDL counter file.
2. A printout of your ModelSim waveforms.
3. Worksheet with tutorial section filled out

Next, recall how you added a set of predefined locations into your UCF file? Now you will find out what those pins do. Open up the following Acrobat file: [dxc_r_RM.pdf](#)

This is the manual for the XCR board. Find Table 3 on the sixth page and draw up a table in your report listing what each pin in your UCF file does, and where it goes to. Keep this file handy, as you will need to use it in the lab itself.

Appendix – VHDL files for download

```

-----
-- XCR7SEG.VHD -- Digilab XCR Seven-Segment Driver
-----
-- Author: James Hansen
-- This file provided for educational purposes only.
-----
-- This module is a circuit to drive the multiplexed seven-segment
-- display found on the Digilab XCR boards.
--
-- Inputs:
-- mclk- system clock
-- dig_l_H- Least significant digit.
-- dig_u_H- Most significant digit.
--
-- Outputs:
-- an- anode lines for the 7-seg displays on DIO1
-- ssg_out_H- cathodes (segment lines) for display on DIO1
--
-----
-- Revision History:
-- 07/03/2002(JamesH): created
-- 03/21/2004(Alonzo Vera): comments and update ISE 5.2
-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity xcr7seg is
Port (
-- Inputs :

    mclk : in std_logic;
    -- clock signal
    dig_l_H : in std_logic_vector(3 downto 0);
    dig_u_H : in std_logic_vector(3 downto 0);

-- Outputs:

    anode : out std_logic_vector(1 downto 0);
    ssg_out_H : out std_logic_vector(6 downto 0)
);
end xcr7seg;

architecture Behavioral of xcr7seg is
-----
-- Component Declarations
-----

-----
-- Signal Declarations
-----

signal clkdiv : std_logic_vector(7 downto 0);
signal cntr : std_logic_vector(3 downto 0);
signal cclk : std_logic;
signal an_sel : std_logic_vector(1 downto 0);
signal an_next : std_logic_vector(1 downto 0);
signal dig : std_logic_vector(6 downto 0);

signal divclk : std_logic;

```

```

signal intclk : std_logic;
signal in_low : std_logic_vector(3 downto 0);
signal in_high : std_logic_vector(3 downto 0);
signal anode_out : std_logic_vector(1 downto 0);
signal display_out : std_logic_vector(6 downto 0);

type state_type is (A, B, C);
signal state, next_state : state_type;

-----
-- Module Implementation
-----

begin

-- Divide the master clock (25.175Mhz) down to a lower frequency.
-- Though with most of the boards, the oscillator has already been
-- dialed down to a very low frequency. As a result, we will
-- just be using the raw system clock for our work.

process (mclk)
begin
    if mclk = '1' and mclk'Event then
        clkdiv <= clkdiv + 1;
    end if;
end process;

-- Decade counter driven by the above clock divider to generate
-- the digits to go out to the seven segment displays.

divclk <= clkdiv(7);

in_low <= dig_l_H;
in_high <= dig_u_H;

-- Next we drive output of the seven-segment display.

display_select : process (mclk)
begin
    if mclk = '1' and mclk'Event then
        an_sel <= an_next;
    end if;
end process;

-- The previous process and the next case-statement determine
-- which seven-segment display is enabled in any given clock
-- cycle.

which_display : process (an_sel)
begin

case an_sel is
    when "01" => an_next <= "10";
    when "10" => an_next <= "01";
    when others => an_next <= "01";
end case;

end process;

drive_display : process (an_sel, in_low, in_high)

begin

case an_sel is

```

```

        when "01" => cntr <= in_low;
        when "10" => cntr <= in_high;
        when others => cntr <= in_high;
    end case;

end process;

-- Now we can display.

anode <= an_sel;
ssg_out_H <= dig;

dig <=
    "0111111" when cntr = "0000" else
    "0000110" when cntr = "0001" else
    "1011011" when cntr = "0010" else
    "1001111" when cntr = "0011" else
    "1100110" when cntr = "0100" else
    "1101101" when cntr = "0101" else
    "1111101" when cntr = "0110" else
    "0000111" when cntr = "0111" else
    "1111111" when cntr = "1000" else
    "1101111" when cntr = "1001" else
    "1110111" when cntr = "1010" else
    "1111100" when cntr = "1011" else
    "0111001" when cntr = "1100" else
    "1011110" when cntr = "1101" else
    "1111001" when cntr = "1110" else
    "1110001" when cntr = "1111" else
    "0000000";

end Behavioral;

```

```

-----
-- TOP_TB.VHD -- Lab 4 Tutorial ModelSim Testbench.
-----
-- Author: <Your name goes here>
-----
-- This is the simulation testbench file for the Lab 4
-- tutorial. The VHDL you will write for the lab
-- itself should be similarly commented. This tells you
-- at a glance what file you're dealing with.
--
-- Inputs: <Your inputs go here.>
-- Outputs: <Your outputs go here>
--
-----
-- Revision History:
-- <Date><Your name>): Created.
-----
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;
USE IEEE.std_logic_unsigned.all;

LIBRARY UNISIM;
LIBRARY XILINXCORELIB;

LIBRARY ieee;
USE IEEE.STD_LOGIC_TEXTIO.ALL;
USE STD.TEXTIO.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE testbench_arch OF testbench IS

COMPONENT TOP
PORT (
    sys_clk : in std_logic; -- System clock.
    count_clk : in std_logic; -- Counter clock.
    in_sw0_h : in std_logic; -- Input switch 1.
    in_sw1_h : in std_logic; -- Input switch 2.

    anodes_out : out std_logic_vector(1 downto 0); -- Anode enables.
    disp_out : out std_logic_vector(6 downto 0) -- Display out.
);
END COMPONENT;

-- This is the counter itself. If you want to test this
-- then uncomment all the XCR_COUNT lines in the VHDL file.
-- COMPONENT XCR_COUNT
-- PORT (
--     sclk_h : in std_logic; -- System clock.
--     sw0_h : in std_logic; -- Switch to control the counter.
--     sw1_h : in std_logic; -- Another switch to control the counter.
--
--     data_out_h : out std_logic_vector(3 downto 0) -- For ModelSim
simulator.
-- );
-- END COMPONENT;

SIGNAL sysc : std_logic; -- Clock signal.
SIGNAL countc : std_logic; -- Secondary clock signal.
SIGNAL in0_h : std_logic; -- An input signal.
SIGNAL in1_h : std_logic; -- A second input signal.
SIGNAL an0_h : std_logic_vector(1 downto 0); -- Output signal.

```

```

SIGNAL disp_h : std_logic_vector(6 downto 0); -- Output signal.

-- SIGNAL out_h : std_logic_vector(3 downto 0); -- If testing just the
-- counter,
-- uncomment this signal and comment out disp_h.

constant CLK_PERIOD : time:= 20 ns;

BEGIN

-- If you wish to test the counter itself, merely comment out this
-- block,
-- and un-comment the XCR_COUNT block. No other changes need
-- to be made to the testbench.

UUT : TOP -- This is the top-level design.

PORT MAP (
    sys_clk => sysc,
    count_clk => countc,
    in_sw0_h => in0_h,
    in_sw1_h => in1_h,
    anodes_out => an0_h,
    disp_out => disp_h
);

-- This is the counter itself. If you want to test this
-- then uncomment all the XCR_COUNT lines in the VHDL file.
-- UUT : XCR_COUNT
-- PORT MAP (
--     sclk_h => sysc,
--     count_clk => countc,
--     sw0_h => in0_h,
--     sw1_h => in1_h,
--     anodes_out => an0_h,
--     data_out_h => out_h
-- );

--Clock for A

CLOCK: process
begin
    sysc <= '0';
    wait for CLK_PERIOD/2;
    sysc <= '1';
    wait for CLK_PERIOD/2;
end process;

-- Secondary count clock. Not always used though.

CLOCK2: process
begin
    countc <= '0';
    wait for CLK_PERIOD;
    countc <= '1';
    wait for CLK_PERIOD;
end process;

-- Tests the counter pause state.

CLOCK3: process
begin
    in0_h <= '1';
    wait for CLK_PERIOD*16;

```

```
        in0_h <= '0';
        wait for CLK_PERIOD*16;
end process;

-- Tests the counter reset state.

CLOCK4: process
begin
    in1_h <= '0';
    wait for CLK_PERIOD*32;
    in1_h <= '1';
    wait for CLK_PERIOD*32;
end process;

END testbench_arch;
```