

Using Concurrent Statements

2021.1

Abstract

This lab illustrates the use of concurrency by building upon the 8-bit register that was designed in the "Using the Tools" lab.

This lab should take approximately 60 minutes.

Objectives

After completing this lab, you will be able to:

- Instantiate a previously created component
- Use concurrent assignments to generate RTL code:
 - Synchronous
 - Combinatorial

Introduction

This lab will build upon the 8-bit register with enable design that was created in the "Using the Tools" lab.

This design is intended for use in the *wave_gen.vhd* module (commonly used in other Xilinx courses) and will steer either data from the UART receiver or the Sample Generator module to the LEDs on the development board. Additional registers are added for the sake of demonstrating instantiation and other concurrent statements.

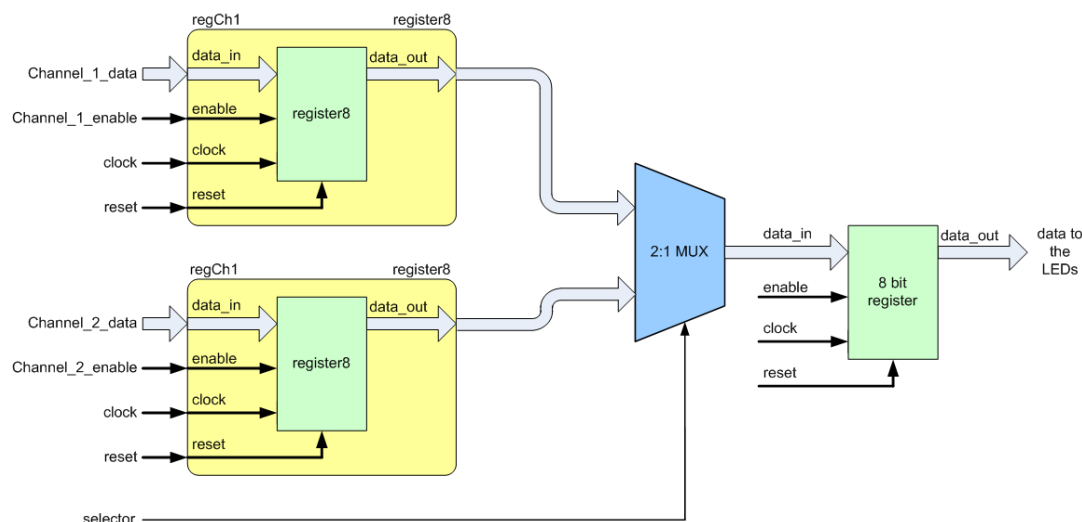


Figure 2-1: Block Diagram for This Lab

Both register8 modules will capture data from both the UART receiver and the Sample Memory module. A selector that identifies which of these two buses is routed through a 2:1 multiplexer to another register (which will be coded using a concurrent conditional assignment) to the LEDs.

While this is not an optimal design, it does illustrate a number of important points:

- Hierarchy (by instantiating the previous lab's register8 component).
- Design re-use (by demonstrating how previously coded modules can be incorporated into a current design).
- Use of conditional concurrent assignments to create a multiplexer (combinatorial) and a register (synchronous).
- Equivalencies in behavior using three different coding styles: instantiation of the register8 (structural), the actual process that implements the functionality of the register8 (behavioral), and the conditional concurrent code (RTL).

Understanding the Lab Environment

The labs and demos provided in this course are designed to run on a Linux platform. Many of the labs and demos can be successfully executed in the Windows environment or a native Linux environment as well.

The instructions found in this lab are expressed using the Linux notation. This includes the forward slash (/) as the hierarchy separator instead of the Windows backslash (\). Students who want to run the labs directly under Windows must use the correct hierarchy separator.

Customizable environment variables enable you to tailor your environment for specific machine configurations. The only environment variable (shown below) used in the customer training environment (CustEd_VM) points to the training directory where all the lab files are located. This reduces the amount of typing you need to do when entering directory paths.

This environment variable can be customized according to your specific location and can be set for Linux systems in the `/etc/profile` file and for Windows systems by entering "env" from the search bar.

The following is the environment variable used in the customer training VM:

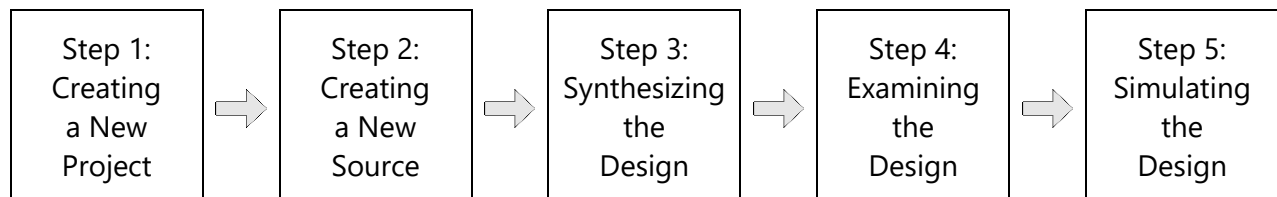
Environment Variable Name	Description
\$TRAINING_PATH	<p>Points to the space allocated for students to work through the labs. This directory includes prebuilt images and starting points for the labs and demos.</p> <p>The customer training VM sets \$TRAINING_PATH to the <code>/home/xilinx/training</code> directory.</p> <p>Typically, Windows users will install the training directory under C: to keep the path names as short as possible.</p>

Note: Environment variables are not supported from the Vitis IDE GUI. When using this tool, you must manually replace `$TRAINING_PATH` with the value of the variable, which in the customer training virtual machine, is `/home/xilinx/training`. Other tools, such as the Vivado Design Suite, will properly expand the environment variable.

Additional note about environments: Both the Vivado Design Suite and Vitis platform offer a Tcl environment. The contents of this environment are NOT preserved with the project. When the tools launch, they start with a pristine Tcl environment with none of the procs or variables remaining from a previous launch of the tools.

This means that if you sourced a Tcl script or manually set any Tcl variables and you closed the tool, when you reopen the tool (and perhaps even reopen the last project), you will need to source the Tcl script again and set any variables that the lab requires.

General Flow



Creating a New Project

Step 1

In this step you will create a new project using the New Project Wizard in the Vivado® Design Suite.

The New Project Wizard in the Vivado IDE will create an XPR project file. The Vivado IDE project file (.xpr) organizes your design files and saves the design status whenever the processes are run from design entry through implementation to programming the targeted Xilinx device.

Here are two popular mechanisms for opening the Vivado Design Suite.

1-1. Open the Vivado Design Suite.

1-1-1. Click the **Vivado** icon (🚀) from the taskbar.

OR

Open a Linux terminal window (press <Ctrl + Alt + T>) and enter the following command:

```
[host]$ source /opt/Xilinx/Vivado/2021.1/settings64.sh
```

```
[host]$ vivado
```

Note: The customer training environment (CustEd_VM) sets the Vivado Design Suite install path to /opt/Xilinx/Vivado. If the Vivado Design Suite is installed in a different location in your environment, use that install path.

The Vivado Design Suite opens to the Welcome window. From the Welcome window you can create a new project, open an existing project, or enter Tcl commands directly into the Vivado Design Suite as well as access documentation and examples.

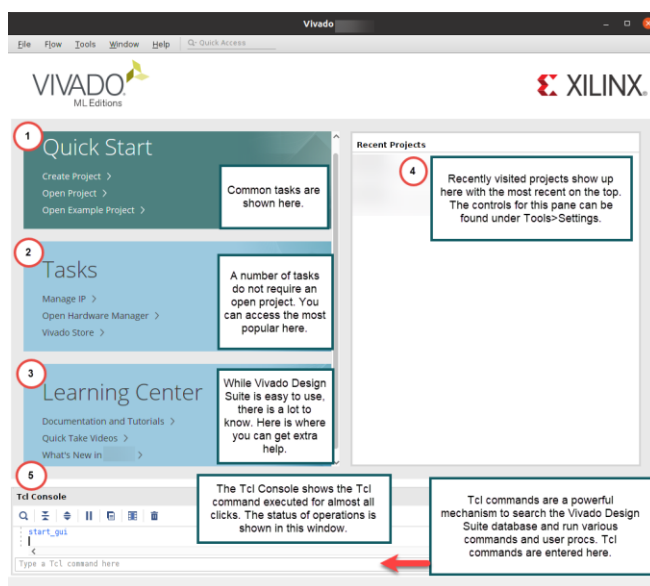


Figure 2-2: Vivado Design Suite Welcome Screen

"Create Project" is the starting point for all designs. Projects contain sources, settings, graphics, IP, and other elements that are used to build a final bitstream and analyze a design. The Create New Project Wizard in the Vivado Design Suite allows you to specify HDL and other project resource files that will be included in the project.

1-2. Create a new, blank Vivado Design Suite project.

1-2-1. Click **Create Project** to begin the process (1).

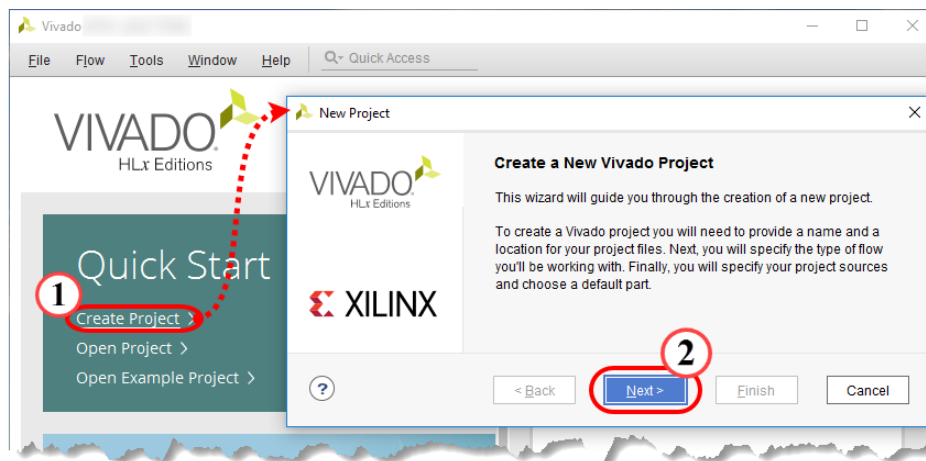


Figure 2-3: Creating a New Vivado Design Suite Project

This will launch the New Project Wizard.

1-2-2. Click **Next** to exit the introductory dialog box and begin entering in project-specific information (2).

1-3. Describe the various aspects of the project.

1-3-1. Enter **concurrency** in the Project name field (1).

1-3-2. Enter the following location in the Project location field (2):

\$TRAINING_PATH/concurrency/lab/KCU105

Important: You need to expand the `path` to its full length as explained in the Introduction section.

Alternatively, you can use the browse feature to navigate to where you want the project to reside.

1-3-3. Deselect the **Create Project Subdirectory** option (3).

Leaving this checked will create an unnecessary level of hierarchy for this lab.

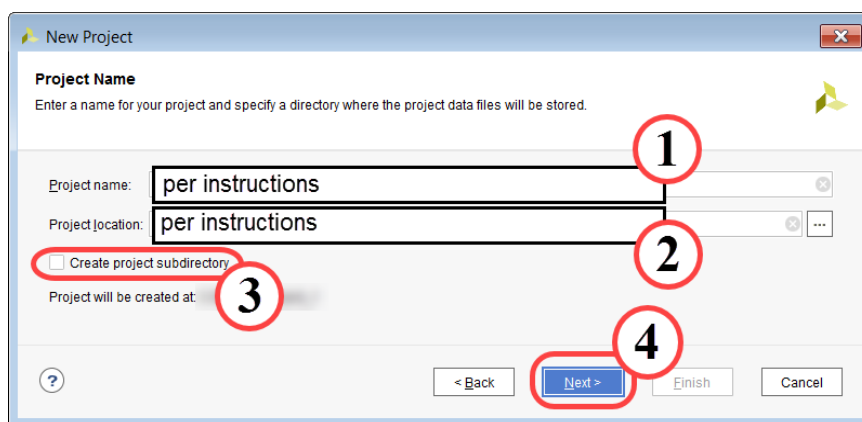


Figure 2-4: Entering the Project Name and Location

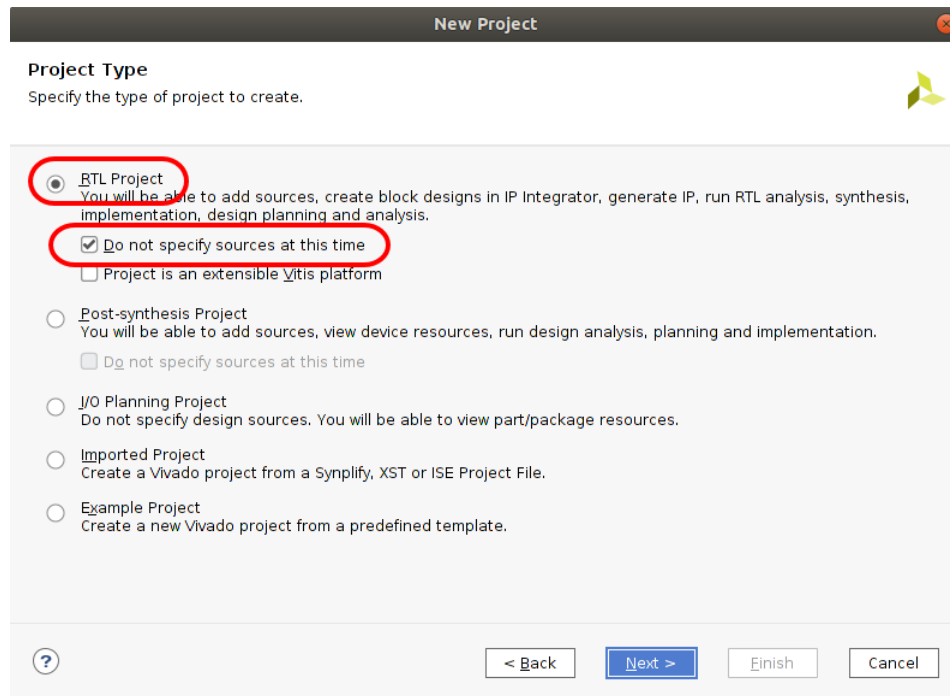
1-3-4. Click **Next** to advance to the next dialog box (4).

Here you will specify your project type as either an RTL project or a post-synthesis project. Simply put, an RTL project enables you to add or create new HDL files and synthesize them, whereas the post-synthesis project requires pre-synthesized files. When an empty design is created, an RTL project is used.

1-3-5. Select **RTL Project**.

1-3-6. Select **Do not specify sources at this time**, which creates a blank project.

While existing sources could be entered at this time, you will enter them later so that you can move through this portion of the project creation process more quickly.



New Project

Project Type
Specify the type of project to create.

☒ **RTL Project**
You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.

☒ **Do not specify sources at this time**

☐ Project is an extensible **Vitis** platform

☐ **Post-synthesis Project**
You will be able to add sources, view device resources, run design analysis, planning and implementation.

☐ Do not specify sources at this time

☐ **I/O Planning Project**
Do not specify design sources. You will be able to view part/package resources.

☐ **Imported Project**
Create a Vivado project from a Synplify, XST or ISE Project File.

☐ **Example Project**
Create a new Vivado project from a predefined template.

? < Back Next > Finish Cancel

Figure 2-5: Specifying Project Options

1-3-7. Click **Next** to advance to the target device/platform selection.

1-4. Select the target part by first filtering by board and then by family. If you are not using a supported board, you will need to filter by part.

1-4-1. Click **Boards** from the *Default part* area to filter by board rather than by the specific part (1).

1-4-2. Select **xilinx.com** from the Vendor drop-down list in the Filter area (2).

This limits the number of boards seen to those manufactured by the specified vendor.

1-4-3. Select **Kintex-UltraScale KCU105 Evaluation Platform** from the board list.

If you accidentally double-clicked the entry, a web page will open for that board. You can close the browser page.

Note: While this page contains important information and resources for the board, these details are not needed to complete this lab.

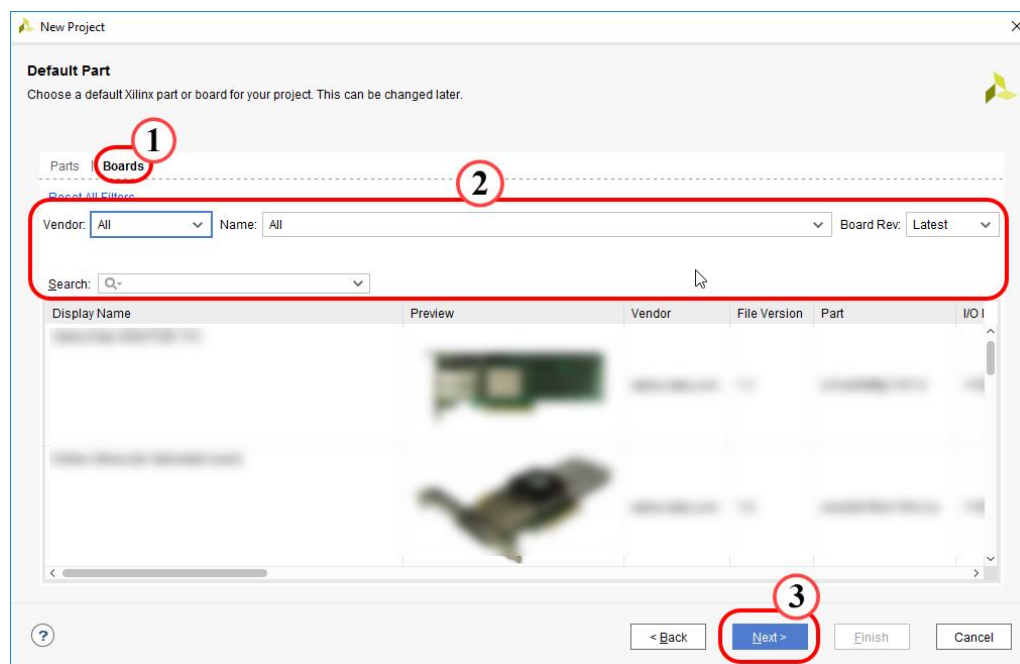


Figure 2-6: Selecting the Board for the Project

1-4-4. Click **Next** to advance to the summary (3).

A summary of your project is displayed. If you want to change any of the information that you entered, you can do so now by clicking **Back** until you reach the correct dialog box. Once the project is created, the project properties can still be edited.

1-4-5. Click **Finish** to accept these settings and build the project.

Your project is constructed and leaves you in the operational portion of the Vivado Design Suite GUI.

1-4-6. Click **Settings** under Project Manager in the Flow Navigator and change the Target Language to **VHDL**.

Creating a New VHDL Source

Step 2

Now that the project has been created, it is time to begin creating the design. You will use the *register8* module from the last lab as well as create a new module that "calls" the *register8* module and adds additional logic for the multiplexer.

Refer to the design block diagram in the Introduction section of this lab.

Now you will code a VHDL module named *LED_manager* that will serve as the driver for the LEDs and contain all the registers and the multiplexer.

2-1. Create a new HDL source file called *LED_manager*.

2-1-1. Select **Add Sources** in the Flow Navigator under Project Manager.

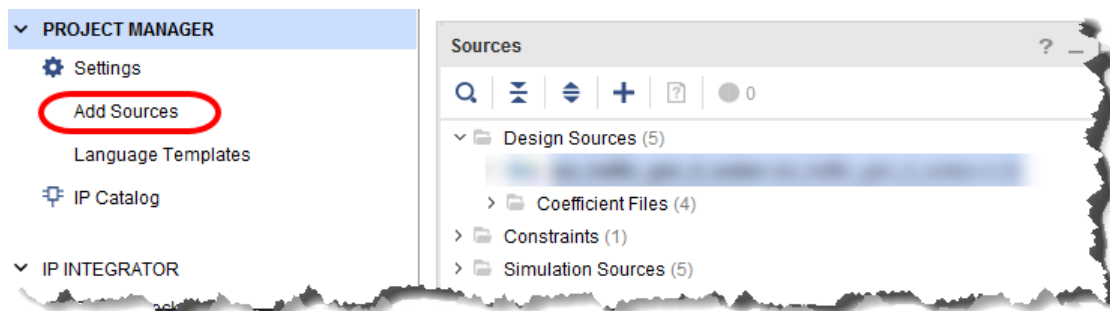


Figure 2-7: Selecting Add Sources

2-1-2. Select **Add or create design sources (1)**.

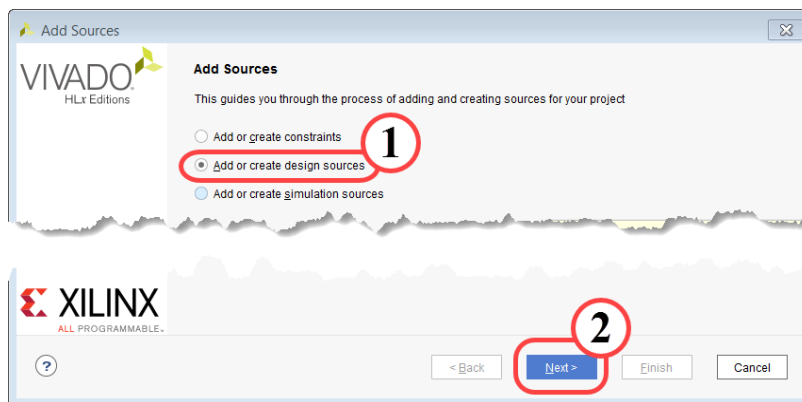


Figure 2-8: Selecting Add or Create Design Sources

2-1-3. Click **Next** (2).

The Add or Create Design Sources dialog box opens.

2-1-4. Click the **Plus (+)** icon and select **Create File**.

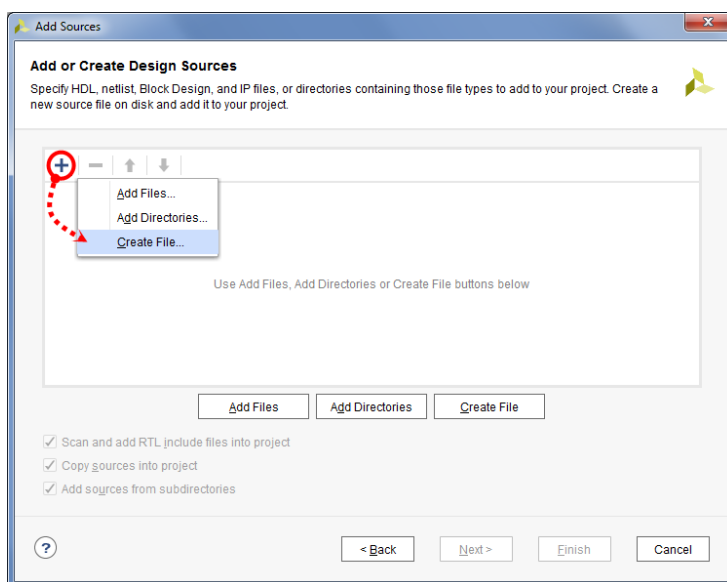


Figure 2-9: Selecting Create File

The Create Source File dialog box opens.

2-1-5. Select **VHDL** from the File type drop-down list.

2-1-6. Enter **LED_manager** as the file name.

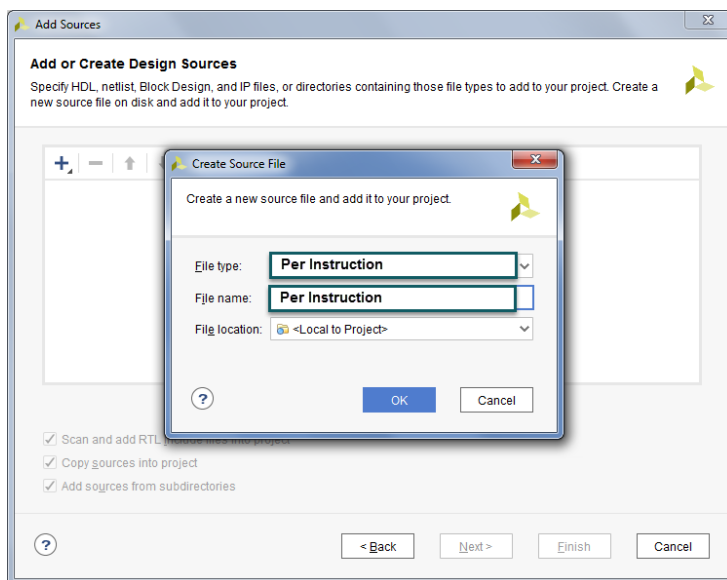


Figure 2-10: Entering File Name and Type

2-1-7. Click **OK** in the Create Source File dialog box.

2-1-8. Click **Finish** to add the new source file(s).

The Define Module dialog box opens.

2-2. Specify the following inputs and outputs for *LED_manager* in the Define Module dialog box.

- **Channel_1_data** – in – std_logic_vector – width 8 (i.e., 7 downto 0)
- **Channel_2_data** – in std_logic_vector – width 8 (i.e., 7 downto 0)
- **Channel_1_enable** – in – std_logic
- **Channel_2_enable** – in – std_logic
- **selector** – in – std_logic
- **clock** – in – std_logic
- **reset** – in – std_logic
- **data_out** – out – std_logic_vector – width 8 (i.e., 7 downto 0)

2-2-1. Fill in the dialog box as shown in the figure below.

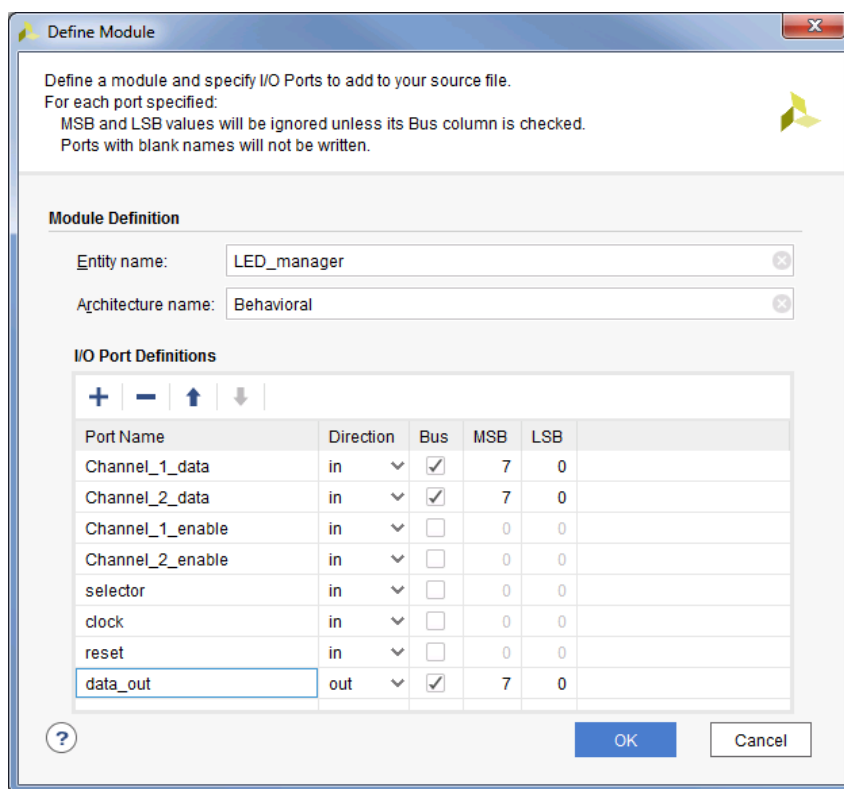


Figure 2-11: Defining Ports for LED_Manager

2-2-2. Click **OK**.

The newly formed code will appear in the Sources view.

2-2-3. Double-click the **LED_manager.vhd** module in the Hierarchy tab of Sources window to launch the file in a text editor window of the Vivado IDE.

2-3. Uncomment the IEEE.NUMERIC_STD package in the code.

The VHDL template automatically includes the IEEE library and uses the **STD_LOGIC_1164** package, which contains the definitions for **std_logic** and **std_logic_vector**.

Although the **NUMERIC_STD** package is not needed for this lab you can uncomment this statement as there are no negative effects.

One more library is provided in the template: the **UNISIM** library. This library contains the package **Vcomponents** and is used when Xilinx primitives such as buffers and clock management resources are used.

Again, none of these primitives are going to be explicitly declared in this file and you can leave this code commented out. If you choose to uncomment this block, there are no negative effects.

- 2-3-1. Remove the `--` comment symbols before **use IEEE.NUMERIC_STD.ALL** on line 27 to include the **NUMERIC_STD** package.

```
21 |
22 | library IEEE;
23 | use IEEE.STD_LOGIC_1164.ALL;
24 |
25 | -- Uncomment the following library declaration if using
26 | -- arithmetic functions with Signed or Unsigned values
27 | --use IEEE.NUMERIC_STD.ALL;
28 |
29 | -- Uncomment the following library declaration if instantiating
30 | -- any Xilinx leaf cells in this code.
31 | --library UNISIM;
32 | --use UNISIM.VComponents.all;
33 |
```

Figure 2-12: Source Code with **NUMERIC_STD** Package Commented

Notice how the line color changes on line 27.

```
22 | library IEEE;
23 | use IEEE.STD_LOGIC_1164.ALL;
24 |
25 | -- Uncomment the following library declaration if using
26 | -- arithmetic functions with Signed or Unsigned values
27 | use IEEE.NUMERIC_STD.ALL;
28 |
29 | -- Uncomment the following library declaration if instantiating
```

Figure 2-13: Current State of the Source

This indicates that this is a predefined library, package, or type defined in a package.

Also notice that the keywords are colored violet. These visual cues act as a quick confirmation that you entered the information as intended.

- 2-3-2. Select **File > Text Editor > Save File**.

2-4. Now that you have defined the inputs and outputs for this module, the behavior must be described.

Because the inputs connect to two instances of the *register8* component, import the *register8.vhd* source into the project.

2-4-1. Click **Add Sources** under Project Manager in the Flow Navigator.

2-4-2. Select **Add or Create Design Sources**.

2-4-3. Click **Next**.

2-4-4. Click the **Plus (+)** icon and select **Add Files**.

2-4-5. Select **register8.vhd** from the location `$TRAINING_PATH/concurrency/support`.

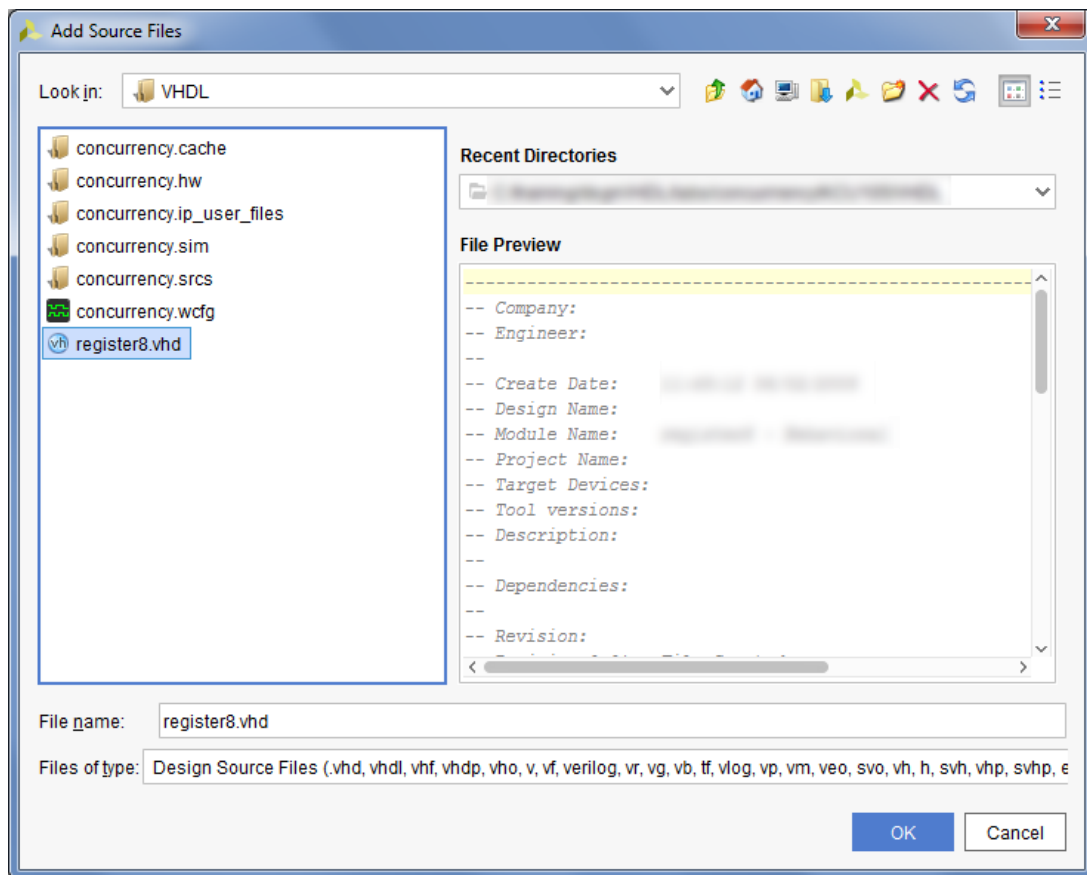


Figure 2-14: Adding register8.vhd to the Project

2-4-6. Click **OK**.

2-4-7. Click **Finish** to close the Add or Create Design Sources dialog box.

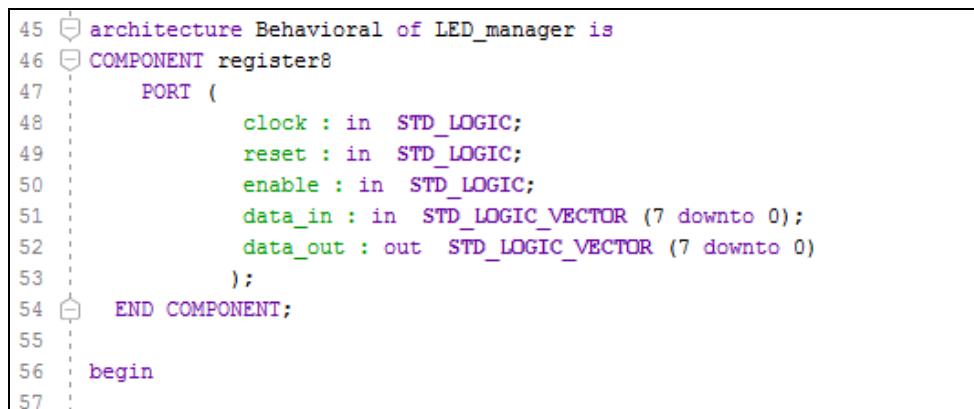
Notice the presence of *register8* at the same level of hierarchy as *LED_manager*. This is because *register8* has not yet been declared as a subordinate of *LED_manager*. Also notice the icon next to *LED_manager*. This icon implies that the file is the "top" module in the design.

2-5. With *register8* now in the project, instantiate it twice into *LED_manager* and connect it to the input pins. Name the instantiations *regCh1* and *regCh2*. Make the following connections:

- **regCh1**
 - **clock:** clock
 - **reset:** reset
 - **enable:** Channel_1_enable
 - **data_in:** Channel_1_data
 - **data_out:** *Leave it open for now; you will make this assignment in another step*
- **regCh2**
 - **clock:** clock
 - **reset:** reset
 - **enable:** Channel_2_enable
 - **data_in:** Channel_2_data
 - **data_out:** *Leave it open for now; you will make this assignment in another step*

2-5-1. Open **LED_manager** in a text editor in the Sources view, if not already opened.

2-5-2. Code the component declaration of *register8* into *LED_manager* between the "architecture" and "begin" statements near line 47.



```
45 architecture Behavioral of LED_manager is
46   COMPONENT register8
47     PORT (
48         clock : in  STD_LOGIC;
49         reset  : in  STD_LOGIC;
50         enable : in  STD_LOGIC;
51         data_in : in  STD_LOGIC_VECTOR (7 downto 0);
52         data_out : out STD_LOGIC_VECTOR (7 downto 0)
53     );
54   END COMPONENT;
55
56   begin
57
```

Figure 2-15: Copying the Component Declaration from the Instantiation Template into *LED_manager*

2-5-3. Code the *register8* instantiation into *LED_manager* as shown in the figure below.

The component instantiations should be placed between the "begin" and "end" statements in the architecture.

2-5-4. Repeat this process so that there are two copies of the instantiation.

Name the modules *regCh1* and *regCh2*, because the inputs have been named as channels, and to keep the modules generic for potential future use.

This is done so that two separate copies of *register8* will exist: one for the received data from the UART and the other for data from the sample memory.

The instantiations must be named uniquely.

2-5-5. Assign the Channel 1 signals to the Channel 1 (*regCh1*) register instantiation to connect the input pins to the inputs of the instantiations.

2-5-6. Repeat for Channel 2 (*regCh2*).

Clock and reset are common to both instantiations.

The code should appear similar to the figure below.

Type text here

```
58  begin
59
60  regCh1: register8 port map(clock => clock,
61                             reset => reset,
62                             enable => Channel_1_enable,
63                             data_in => Channel_1_data,
64                             data_out =>                                     );
65
66  regCh2: register8 port map(clock => clock,
67                             reset => reset,
68                             enable => Channel_2_enable,
69                             data_in => Channel_2_data,
70                             data_out =>                                     );
71
72  end Behavioral;
```

Figure 2-16: Two Instantiations of register8 with Inputs Assigned

- 2-6. Name the outputs of *register8* as *Channel_1_registered_data* and *Channel_2_registered_data* and connect to the multiplexer.

Write the multiplexer as a "with/select" conditional concurrent assignment and create a new signal named *mux_data_selected* to contain the output of the multiplexer.

Remember to define these signals as 8-bit *std_logic_vector*s between the *architecture* and the *begin* statements.

- 2-6-1. Name the two signals *Channel_1_registered_data* and *Channel_2_registered_data* to define them as the outputs from the *register8* instantiations.

```
55
56     signal Channel_1_registered_data : std_logic_vector(7 downto 0);
57     signal Channel_2_registered_data : std_logic_vector(7 downto 0);
58
```

Figure 2-17: Signals for the Outputs of the *register8* Instantiations Defined

- 2-6-2. Assign the outputs of the two *register8* instantiations.

```
58     begin
59     regCh1: register8 port map(clock => clock,
60                               reset => reset,
61                               enable => Channel_1_enable,
62                               data_in => Channel_1_data,
63                               data_out => Channel_1_registered_data );
64
65     regCh2: register8 port map(clock => clock,
66                               reset => reset,
67                               enable => Channel_2_enable,
68                               data_in => Channel_2_data,
69                               data_out => Channel_2_registered_data );
70
71     end Behavioral;
```

Figure 2-18: Registered Outputs for the *register8* Instantiations

Take note of the current file structure in the Hierarchy window.

- 2-6-3. Select **File > Text Editor > Save** to preserve your work so far.

2-7. Code and add the multiplexer after the second instantiation.

You need to declare a signal *mux_data_selected* with *std_logic_vector* type of width 8 (i.e., 7 downto 0).

Question 1

How do you code a multiplexer using a "with/select" assignment?

First you need a select input. Then, you state the conditions that would select the data

2-7-1. Code the multiplexer with a "with/select" statement.

2-8. Finally, the output from the multiplexer (*data_out*) should be registered—it is good practice to register the outputs of each module.

Using the "when" form of a conditional concurrent assignment, code an 8-bit register. The register will clear one clock cycle after reset is applied to the *register8* instantiations.

Question 2

How would you code this using a concurrent "when" assignment?

register8 code already has a reset and process, so I do not have to write a process statement

and reset statement. --Code: *data_out* <= *mux_data_selected* when rising_clk(clock)

2-8-1. Add the concurrent statement from your answer into the code after the multiplexer.

Technically the code can be added anywhere between the architecture's begin and end statement; however, because this design is fairly linear, it makes more organizational sense to add it at the end.

The final code should appear similar to the figure below.

```

44 :
45 architecture Behavioral of LED_manager is
46   COMPONENT register8
47     PORT (
48       clock : in  STD_LOGIC;
49       reset : in  STD_LOGIC;
50       enable : in  STD_LOGIC;
51       data_in : in  STD_LOGIC_VECTOR (7 downto 0);
52       data_out : out  STD_LOGIC_VECTOR (7 downto 0)
53     );
54   END COMPONENT;
55
56   signal Channel_1_registered_data : std_logic_vector(7 downto 0);
57   signal Channel_2_registered_data : std_logic_vector(7 downto 0);
58   signal mux_data_selected : std_logic_vector(7 downto 0);      --combinational output of multiplexer
59
60   begin
61     regCh1: register8 port map(clock => clock,
62                               reset => reset,
63                               enable => Channel_1_enable,
64                               data_in => Channel_1_data,
65                               data_out => Channel_1_registered_data );
66
67     regCh2: register8 port map(clock => clock,
68                               reset => reset,
69                               enable => Channel_2_enable,
70                               data_in => Channel_2_data,
71                               data_out => Channel_2_registered_data );
72
73     --construct a two input multiplexer
74     with selector select
75       mux_data_selected <= Channel_1_registered_data when '0',
76                           Channel_2_registered_data when '1',
77                           (others => '-') when others;
78
79     -- registering output
80     data_out <= mux_data_selected when rising_edge(clock);
81
82
83   end Behavioral;
84

```

Figure 2-19: Final Version of Code

2-8-2. Select **File > Text Editor > Save File**.

Question 3

How did the Hierarchy view change after you saved the file?

Register8 file became apart of the library and regCh1 and regCh2 were nested under

LED_manager

Selecting Synthesis Options and Synthesizing

Step 3

Now that the code has been entered, you must verify that it is what you intended. Because synthesis options can have a profound impact on how code turns into FPGA fabric, you will first set several commonly used synthesis options.

3-1. Set the option to tell the tools to not flatten the hierarchical design.

Although the tools can perform a limited amount of combinatorial logic optimization when the hierarchy is flattened, good coding practice calls for registers at each level of hierarchy, thus removing the synthesis tool's ability to optimize.

The benefits of keeping the hierarchy intact is so that signals can be more readily found when other tools are used, such as the schematic viewers and the simulation tools.

3-1-1. Select **Settings** from the Flow Navigator and select the **Synthesis** tab.

The Synthesis Settings dialog box opens.

3-1-2. Set the **-flatten_hierarchy** option to **none** as shown below.

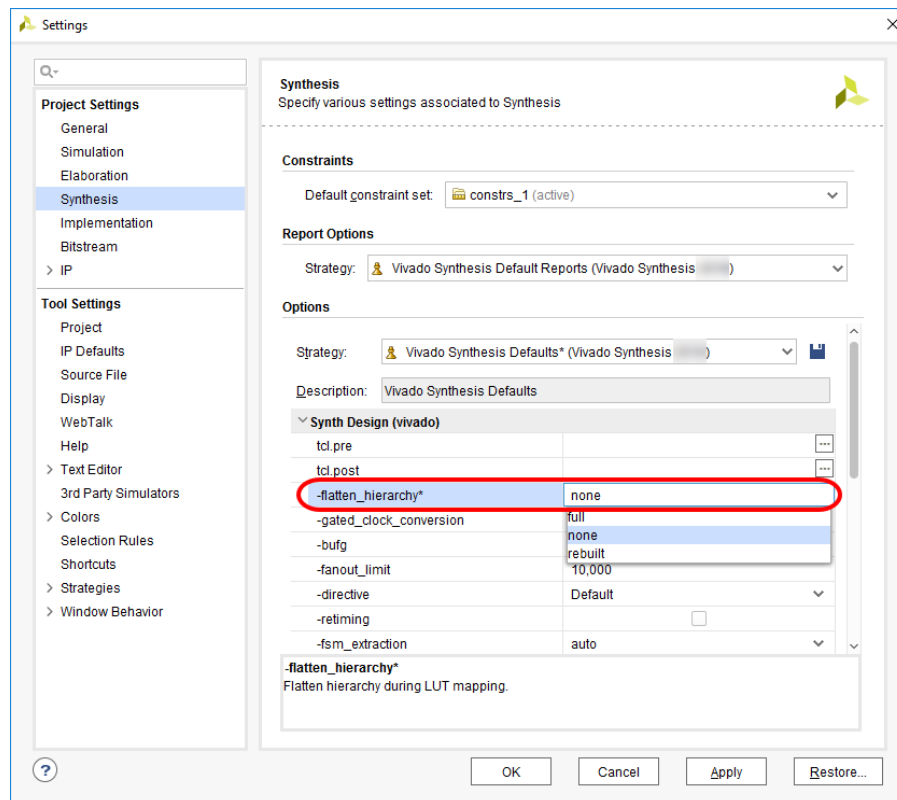


Figure 2-20: Keeping the Hierarchy

3-1-3. Click **OK**.

3-2. Run synthesis.

3-2-1. Click **Run Synthesis** in the Flow Navigator under Synthesis.

Alternatively, you can also select **Flow > Run Synthesis** or press <F11>.

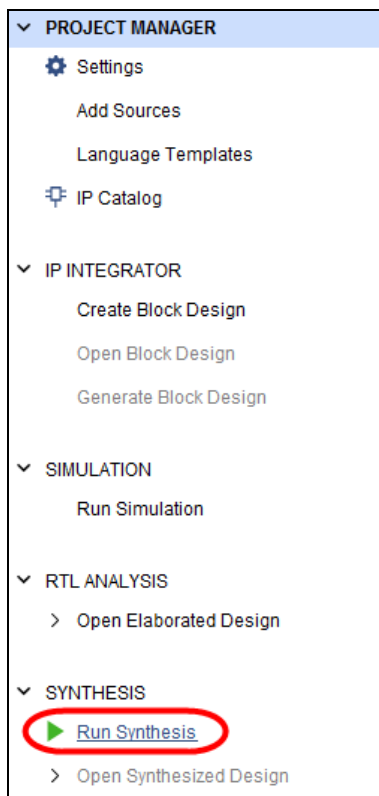


Figure 2-21: Selecting Run Synthesis

The Launch Runs dialog box opens.

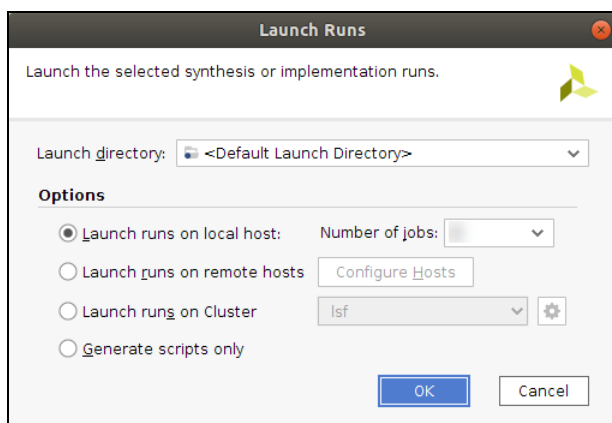


Figure 2-22: Setting the Launch Run Configuration

Hint: When launching the runs on a local host, it is common to set the number of jobs to the maximum value as this recruits the largest number of processors for the task and typically results in the least amount of time spent in synthesis.

3-2-2. Click **OK** to launch the runs.

3-2-3. Click **Save** if you are asked to save your files.

Once synthesis completes, you are asked what task you want to perform next: run implementation, open the synthesized design, view reports, or none of the above.

3-2-4. Select **Open Synthesis Design** (1).

3-2-5. [Optional] If you are familiar with accessing these various capabilities, you can disable this dialog box from appearing again by selecting **Don't show this dialog again** (2).

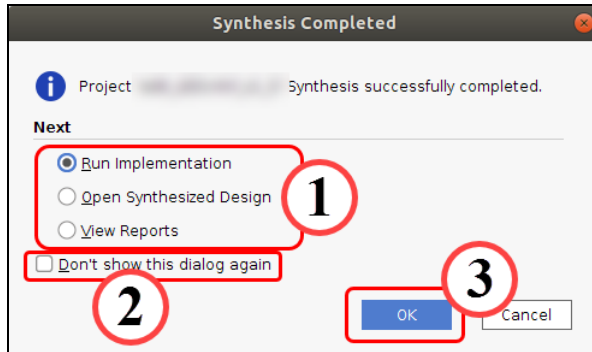


Figure 2-23: Selecting After Synthesis Options

3-2-6. Click **OK** to take the action you just selected (3) or click **Cancel** to simply close the dialog box.

Examining the Design with Schematic Viewer

Step 4

How do you know if the code built the logic that was expected? Simulation would be a good choice and will be covered in the next step. In the meantime, you will view the generated code using the Schematic viewer available in the Vivado Design Suite.

4-1. Use the Schematic viewer to explore the design.

The Schematic viewer shows the logic and register relationships and how they are implemented in the fabric.

The Vivado IDE view changes when the synthesized design is opened. The Package and Device views should be visible.

4-1-1. Open the synthesized design and select the **Netlist** tab to switch to the Netlist view.

4-1-2. Right-click **LED_manager** in the Netlist view and select **Schematic**.

The final design should appear as shown in the figure below, depending on the board you are using.

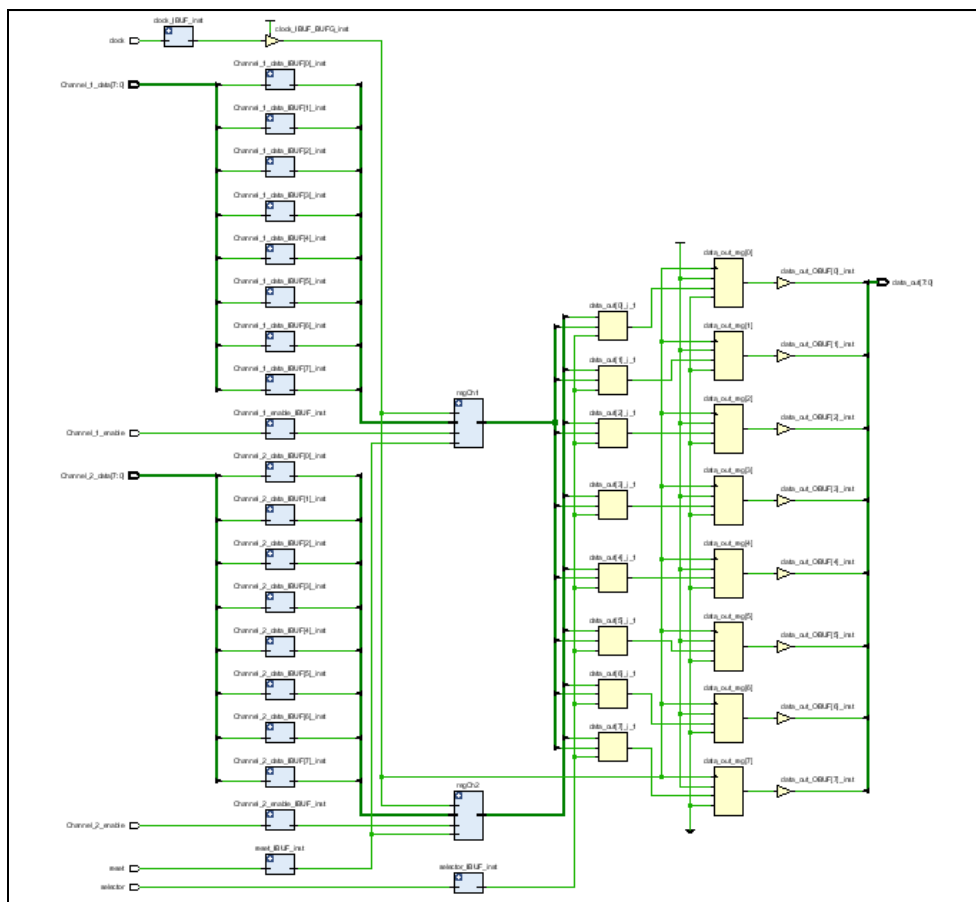


Figure 2-24: Full Schematic View (KCU105)

Simulating the Design

Step 5

Looking at the generated logic is one way to validate that a design has been built the way you wanted; however, this is impractical for large designs. A testbench is a better way because the results of the logic (based on certain stimulus) can be viewed.

A testbench has already been constructed for use with this module. Your task is to apply the testbench to your design and validate the results.

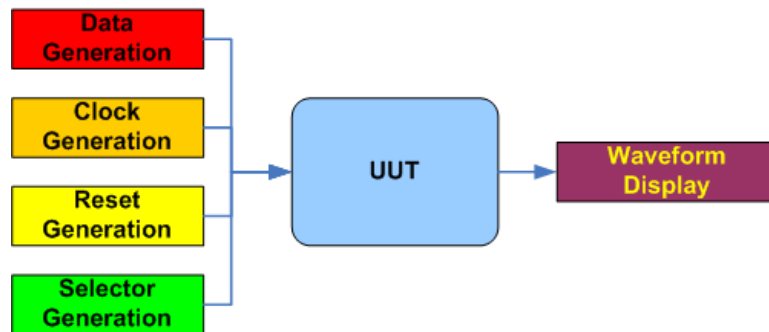


Figure 2-25: Simple Testbench

5-1. Create a testbench with the name *LED_manager_tb* and add it to the project.

5-1-1. Click **Add Sources** in the Flow Navigator under Project Manager.

The Add Sources Wizard opens.

Step 2. Select **Add or create simulation sources**.

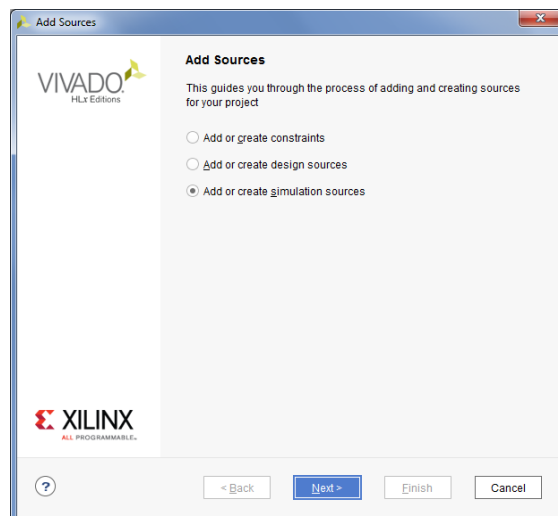


Figure 2-26: Add Sources for Simulation

The Add or Create Simulation Sources dialog box opens.

5-1-3. Click **Next**.

5-1-4. Click the **Plus (+)** icon and select **Create File**.

5-1-5. Select **VHDL** as the file type.

5-1-6. Enter **LED_manager_tb** as the filename and click **OK**.

5-1-7. Click **Finish**.

5-1-8. Click **OK** in the Define module dialog box to create a module without ports.

Note: Testbenches do not include ports because they are meant to test the design rather than be part of the design.

5-1-9. Click **Yes** in the Define module dialog box to confirm creating the module without ports.

Question 4

After *LED_manager_tb* is added, why does it not appear in the Design Sources hierarchy?

It's not apart of the design. It's just used to simulate the design

Write a testbench to simulate the *LED_manager* module.

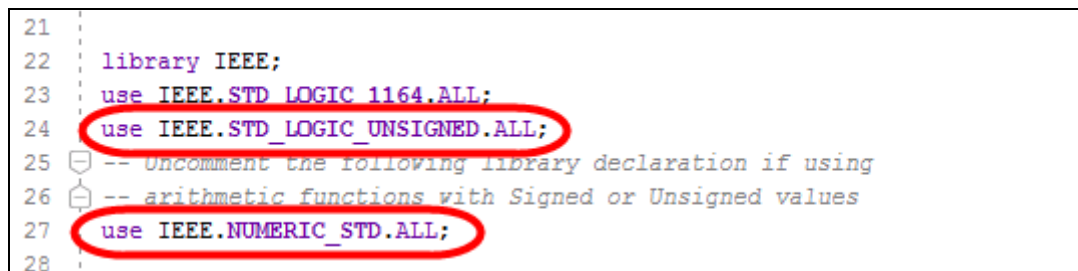
5-2. Include the libraries in *testbench LED_manager_tb.vhd*.

5-2-1. Expand **Simulation Sources** > **sim_1**.

5-2-2. Double-click the **LED_manager_tb.vhd** file.

5-2-3. Include the *use IEEE.STD_LOGIC_UNSIGNED.ALL;* library as shown in the figure below.

5-2-4. Uncomment line *use IEEE.NUMERIC_STD.ALL;* as shown in the figure below.



```
21
22 library IEEE;
23 use IEEE.STD_LOGIC_1164.ALL;
24 use IEEE.STD_LOGIC_UNSIGNED.ALL;
25 -- Uncomment the following library declaration if using
26 -- arithmetic functions with Signed or Unsigned values
27 use IEEE.NUMERIC_STD.ALL;
28
```

Figure 2-27: Including the Libraries in the Testbench (LED_manager_tb.vhd)

5-3. Verify the architecture for the testbench.

5-3-1. Verify the architecture for the testbench as shown in the figure below.

```

22  library IEEE;
23  use IEEE.STD_LOGIC_1164.ALL;
24  use IEEE.STD_LOGIC_UNSIGNED.ALL;
25  -- Uncomment the following library declaration if using
26  -- arithmetic functions with Signed or Unsigned values
27  use IEEE.NUMERIC_STD.ALL;
28
29  -- Uncomment the following library declaration if instantiating
30  -- any Xilinx leaf cells in this code.
31  --library UNISIM;
32  --use UNISIM.VComponents.all;
33
34  entity LED_manager_tb is
35  -- Port ( );
36  end LED_manager_tb;
37
38  architecture Behavioral of LED_manager_tb is
39
40  begin
41
42
43  end Behavioral;
44

```

Figure 2-28: Architecture for testbench LED_manager_tb.vhd

5-4. Declare a component declaration for the *LED_manager* module (unit under test).

5-4-1. Declare a component declaration for the *LED_manager* module (UUT) between the ARCHITECTURE and BEGIN statements as shown below.

```

38  architecture Behavioral of LED_manager_tb is
39
40  COMPONENT LED_manager
41  PORT(
42      Channel_1_data  : IN  std_logic_vector(7 downto 0);
43      Channel_2_data  : IN  std_logic_vector(7 downto 0);
44      Channel_1_enable : IN  std_logic;
45      Channel_2_enable : IN  std_logic;
46      Selector        : IN  std_logic;
47      clock            : IN  std_logic;
48      reset            : IN  std_logic;
49      data_out         : OUT std_logic_vector(7 downto 0)
50  );
51  END COMPONENT;
52  begin
53
54
55  end Behavioral;
56

```

Figure 2-29: Component Declaration of the UUT

5-5. Instantiate the *LED_manager* module (UUT).

5-5-1. Instantiate the *LED_manager* module below the architecture's BEGIN statement as shown in the figure below.

```

52
53 begin
54   -- Instantiate the Unit Under Test (UUT)
55   uut: LED_manager PORT MAP (
56       Channel_1_data => ,
57       Channel_2_data => ,
58       Channel_1_enable => ,
59       Channel_2_enable => ,
60       Selector       => ,
61       clock          => ,
62       reset          => ,
63       data_out       =>
64   );
65
66   end Behavioral;
67

```

Figure 2-30: Instantiating the UUT

This is the module that you will test by providing stimulus to the input ports and verifying the outputs.

5-6. Declare the input and output signals and connect to the UUT.

5-6-1. Declare the input and output signals between the ARCHITECTURE and BEGIN statements.

5-6-2. Connect the UUT with the declared signals as shown in the figure below.

```

54   --Inputs
55   signal Channel_1_data : std_logic_vector(7 downto 0) := (others => '0');
56   signal Channel_2_data : std_logic_vector(7 downto 0) := (others => '0');
57   signal Channel_1_enable : std_logic := '0';
58   signal Channel_2_enable : std_logic := '0';
59   signal Selector       : std_logic := '0';
60   signal clock          : std_logic := '0';
61   signal reset          : std_logic := '0';
62
63   --Outputs
64   signal data_out : std_logic_vector(7 downto 0);
65
66 begin
67   -- Instantiate the Unit Under Test (UUT)
68   uut: LED_manager PORT MAP (
69       Channel_1_data => Channel_1_data,
70       Channel_2_data => Channel_2_data,
71       Channel_1_enable => Channel_1_enable,
72       Channel_2_enable => Channel_2_enable,
73       Selector       => Selector,
74       clock          => clock,
75       reset          => reset,
76       data_out       => data_out
77   );
78   end Behavioral;

```

Figure 2-31: Declaring the Input and Output Signals and Assigning to the UUT

5-6-3. Save the file.

The UUT *LED_manager* module is now under the hierarchy of the testbench *LED_manager_tb*.

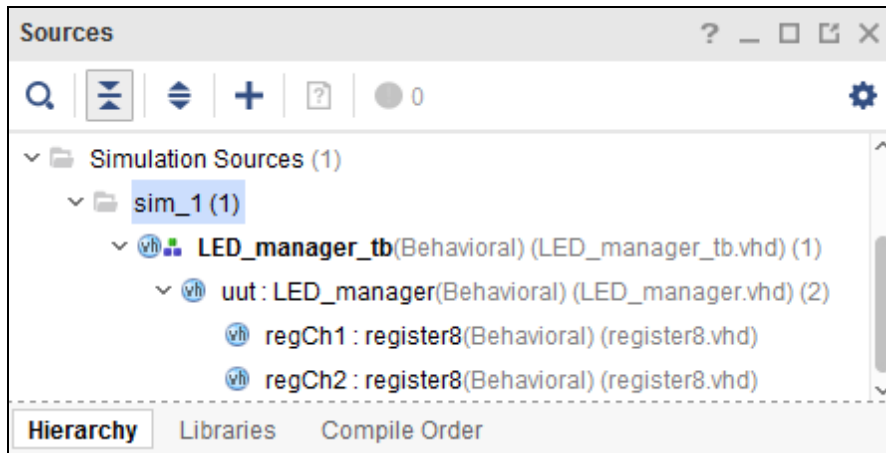


Figure 2-32: UUT Under the Hierarchy of testbench *LED_manager_tb*

5-7. Generate a stimulus for reset and clock.

5-7-1. Declare the constant for clock_period of 10 ns between the ARCHITECTURE and BEGIN statements.

```

63      --Outputs
64      signal data_out : std_logic_vector(7 downto 0);
65
66      -- Clock period definitions
67      constant clock_period : time := 10 ns;
68

```

Figure 2-33: Constant Declaration for Time Period

5-7-2. Write code to reset (active HIGH) the signal by making it ten times the clock period and then deasserting.

5-7-3. Write code to generate a clock for a period of 5 ns between the begin and UUT instantiation.

```

82      -- reset generation
83      reset <= '1', '0' after clock_period * 10;
84
85      -- clock generation
86      clock <= not clock after 5 ns;
87

```

Figure 2-34: Generating Reset and Clock Stimulus

5-8. Generate stimulus for the selector signal.

5-8-1. Write code to generate stimulus for the selector signal as follows:

- For the first 20 times of the clock period, the selector signal should be '0', then becomes '1.'
- Then after 50 times of the clock period, the selector signal should be '0.'
- Finally after 100 times of the clock period, the selector signal should be '1.'

```
82      -- reset generation
83      reset <= '1', '0' after clock_period * 10;
84
85      -- clock generation
86      clock <= not clock after 5 ns;
87
88      -- selector generation
89      selector <= '0', '1' after clock_period * 20, '0' after clock_period * 50, '1' after clock_period * 100;
90
```

Figure 2-35: Generating Stimulus for the Selector Signal

5-9. Generate stimulus for data and enable signals for channel_1 and channel_2.

5-9-1. Add the previous channel_1_data with 0x03 and assign to the channel_1_data signal after 2 times of the clock period.

5-9-2. Add the previous channel_2_data with 0x04 and assign to the channel_2_data signal after 3 times of the clock period.

5-9-3. Enable the channel_1_enable and channel_2_enable signals.

```
91      -- data generation
92      Channel_1_data <= channel_1_data + X"03" after clock_period * 2;
93      Channel_2_data <= channel_2_data + X"04" after clock_period * 3;
94      Channel_1_enable <= '1';
95      Channel_2_enable <= '1';
96
97  end Behavioral;
```

Figure 2-36: Generating Stimulus for channel1 and channel2 Data and Enabling Signals

5-9-4. Save the file.

5-10. Launch the simulation for *LED_manager_tb*.

5-10-1. Select **Simulation** > **Run Simulation** in the Flow Navigator.

5-10-2. Select **Run Behavioral Simulation**.

5-11. Load the predefined simulation settings provided in *concurrency.wcfg* and rerun the simulation for 3 μ s.


For ease of viewing, the predefined simulation settings set the channel_1 information to light blue, the channel_2 information to magenta, and all the buses to unsigned decimal.

5-11-1. Select **File > Simulation Waveform > Open Configuration**.

5-11-2. Browse to the `$TRAINING_PATH/concurrency/support` directory.

5-11-3. Select **concurrency.wcfg** and click **OK**.


5-12. Run the simulation for 3 μ s.


5-12-1. Select **Run > Restart** (or click the  icon) to reset the simulation.

5-12-2. Enter **3 us** in the duration field to set the simulator duration.



Figure 2-37: Vivado Simulator Control Toolbar

5-12-3. Click the  icon to run the simulation for 3 us.

5-12-4. Click the **Zoom Fit**  icon in the waveform toolbar to view the full simulation waveform.

5-12-5. Zoom in using the  icon to analyze the signals.

Question 5

Is the design working? How can you tell?

Yes. The mux select bit is selecting the channels as expected. When 0, it selects

Channel 1, and Channel 2 when 1. The sim file is behaving as coded as well

5-13. Close the simulation.

5-13-1. Select **File > Close Simulation**.

5-14. Close the Vivado Design Suite.

5-14-1. Select **File** > **Exit**.

The Exit Vivado dialog box opens.

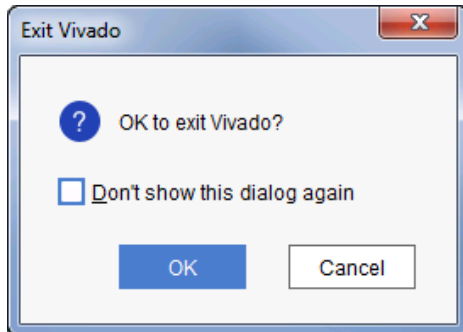


Figure 2-38: Exit Vivado Dialog Box

5-14-2. Click **OK**.

Some systems (particularly VMs) may be memory constrained. Removing the workspace frees a portion of the disk space, allowing other labs to be performed.

You can delete the directory containing the lab you just ran by using the graphical interface or the command line interface. You can choose either mechanism. Both processes will recursively delete all the files in the `$TRAINING_PATH/concurrency` directory.

5-15. [Optional] [Only for local VMs—not for CloudShare] Clean up the file system.

Using the GUI:

5-15-1. Using the graphical browser (Windows: press the <**Windows**> key + <**E**>; Linux: press <**Ctrl** + **N**>), navigate to `$TRAINING_PATH/concurrency`.

5-15-2. Select **concurrency**.

5-15-3. Press <**Delete**>.

-- OR --

Using the command line:

5-15-4. Open a terminal window (Windows: press the <**Windows**> key + <**R**>, then enter **cmd**; Linux: press <**Ctrl** + **Alt** + **T**>).

5-15-5. Enter the following command to delete the contents of the workspace:

[Windows users]: `rd /s /q $TRAINING_PATH/concurrency`

[Linux users]: `rm -rf $TRAINING_PATH/concurrency`

Summary

This lab walked you through the process of instantiating a previously constructed component—the first step in building larger hierarchies (structural coding). You also practiced writing both combinatorial and synchronous concurrent statements (RTL style coding).

Answers

1. How do you code a multiplexer using a "with/select" assignment?

The "with/sel" conditional concurrent construct is roughly equivalent to a "case" statement. It selects one source to be routed to the output per some selection criteria. This is exactly the situation that is needed here to implement a multiplexer.

The "Selector" input provides the item to be selected on and the output must be a bus-wide signal. There are two input buses provided by the output of the each of the register8s.

with Selector select

```
mux_data_selected <= Channel_1_registered_data when '0',
                    Channel_2_registered_data when '1',
                    (others=>'-' ) when others;
```

Note the use of the "others" statement. This is not synthesized; however, with the Selector being a std_logic type, there are nine possible combinations of which only two are specified. Therefore, some kind of "catch all" is needed. This will help clarify the behavior of this design during simulation—that is, if it is seen that mux_data_selected is at value "unknown" then you know that the selection has somehow not been initialized properly and the underlying error will be found more easily.

Remember that mux_data_selected must be declared as a std_logic_vector(7 downto 0).

2. How would you code this using a concurrent "when" assignment?

The data from the multiplexer should be registered on the rising edge of the clock. The line of code that supports this is:

```
data_out <= mux_data_selected when rising_edge(clock);
```

3. How did the Hierarchy view change after you saved the file?

The Vivado IDE scanned the file and determined that the two instantiations of register8 were subordinate to LED_manager module.

4. After *LED_manager_tb* is added, why does it not appear in the Design Sources hierarchy?

The file is for simulation only. You can find it by expanding **Simulation-Only Sources** in the Sources view.

5. Is the design working? How can you tell?

Yes, the design is working.

You can verify by zooming to nearly 200 ns via **View > Zoom In** and scrolling to the left. Repeat this process as many times as you need to in order to see the data values.

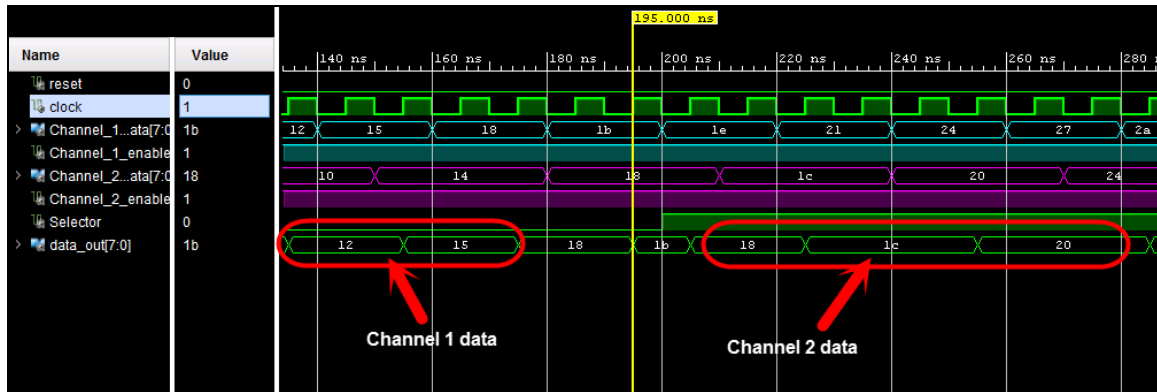


Figure 2-39: Waveform View Zoomed in to See the Values

Notice that Channel_1_data is incrementing by 3 every other clock cycle and channel_2_data is incrementing by 4 every third clock cycle. This shows that the testbench is behaving as expected.

With the selector set in the '0' position, you should see the values from channel 1 on the output (delayed due to the clock registers).

When the selector changes value, notice that on the next clock edge the value drops from 27 to 24, which is the value of the Channel_2_data. This is far from a complete and thorough testbench, but it gives enough exercise to the design to indicate that it is working.