Seminar

Designing with VHDL

Labs

Lecturer

Nir Balulu

Designing with VHDL
Lab Workbook

lang-vhdl-2019.1-wkb-lab-rev1

Designing with VHDL Lab Workbook 2019.1

© Copyright 2019 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, UltraScale, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.

DISCLAIMER

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of the Limited Warranties which can be viewed at http://www.xilinx.com/warranty.htm; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in Critical Applications: http://www.xilinx.com/warranty.htm#critapps. All other trademarks are the property of their respective owners.

Lab FAQ

- Where can I get the files for the labs?
  - www.xilinx.com/training/downloads.htm
  - These are original files and do not contain any work that you may have performed.
  - Labs were developed using version 2019.1 of the tools. Later versions may work and will likely require you to update various pieces of IP. See the "Updating IP" topic in the *Lab Reference Guide* for instructions on how to update IP (Vivado Design Suite Operations > Vivado IP Integrator Operations > Updating IP).
- What if I cannot answer a question in the lab?
  - Do your best! The questions are meant to stimulate thought, not to test your knowledge. After you have pondered a question you can find the answer at the end of each lab.
- Where can I get more detailed information on a topic?
  - The *Lab Reference Guide* is a collection of "how to" topics for commonly performed tasks categorized by tool (Vivado Design Suite, Vivado Analyzer, SDK, etc.) and subdivided into major areas within the tool.
  - The *Lab Reference Guide* is available from the lab files download as well as from www.xilinx.com/training/downloads.htm.
- How do the instructions work?
  - The instructions are provided in layers:

- Steps – these are the major/broadest aspects of solving the problem that the lab poses (X).
- Instructions – these represent the significant instructions towards solving the issue outlined by the step (X-X).
- Tasks – these are the finest granularity items that (when combined with the other tasks) solve the instruction they are subordinate to. Every task is a "clickable" event (X-X-X).

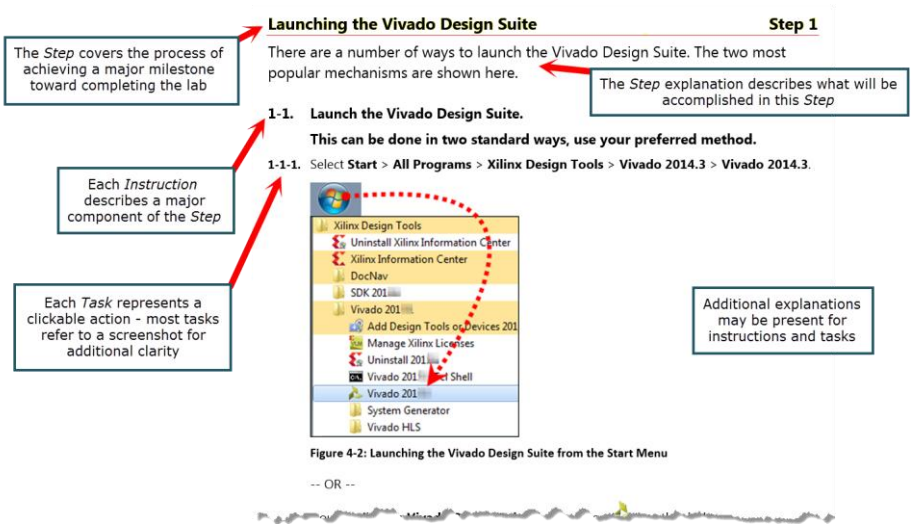


Table of Contents

Lab 1: Using the Tools **3**

Lab 2: Using Concurrent Statements **23**

Lab 3: Designing a Simple Process **55**

Lab 4: Building a Dual-Port Memory **79**

Lab 5: Using Loops **97**

Lab 6: Simulating a Simple Design **121**

Lab 7: Using Functions **139**

Lab 8: Building Your Own Package **151**

Lab 9: Building a Moore Finite State Machine **167**

Lab 10: Building a Meaningful Testbench **191**

Lab 11: Xilinx Tool Flow **207**

© Copyright 2019 Xilinx

Lab 1: Using the Tools

2019.1

Abstract

This lab introduces the VHDL development environment within the Vivado® Design Suite.

This lab should take approximately 45 minutes.

Objectives

After completing this lab, you will be able to:

- Launch and navigate the RTL and technology schematic viewers
- Locate and insert snippets from the Language Templates
- Demonstrate the capabilities of the text editor

Introduction

This lab will introduce the bulk of the features required to successfully complete the Designing with VHDL course labs.

A typical FPGA RTL design process is as follows:

1. Create new project.
2. Add/create design source files.
3. Integrate systems (embedded or DSP).
4. Add IP.
5. Elaboration of RTL.
6. RTL simulation.
7. RTL analysis.
8. Evaluate clocking resources.
9. Plan I/O pin layout.
10. Run DRC checker.
11. Synthesize.
12. Add timing constraints and check timing.
13. Re-synthesize.
14. Review design reports.
15. Implement.
16. Review design reports and perform static timing analysis.
17. Evaluate design with routing resources and/or with the Vivado logic analyzer.

In order to get their design to work properly, many designers would say that they need to backtrack in this process and repeat certain steps. Many designers would also lock their pins earlier in the design flow. This flow is to illustrate what the lab is leading to.

Here you will use the Vivado Design Suite to create a new design project, create a new VHDL design source file using the Language Templates provided in the Vivado Design Suite (i.e.,

© Copyright 2019 Xilinx

design examples), synthesize the design by using Vivado synthesis, and analyze the synthesized schematic.

The circuit designed in this lab is a simple 8-bit register. You will use the VHDL code of a specific type of D flip-flop, provided in the Language Templates, to complete this design.

Note: Do not concern yourself with the specific VHDL statements—they will be covered in great detail during the course.

Understanding the Lab Environment

The labs and demos provided in this course are designed to run under the provided Linux platform. The provided Linux platform has a customized lab environment. These customizations have been made to simplify and enhance your learning experience. Many labs and demos can be successfully executed in the Windows environment as well. Topics including PetaLinux, Yocto, and QEMU are generally not supported under Windows.

The instructions, for the sake of brevity, are expressed using the Linux notation. This includes the forward slash ('/') as the hierarchy separator instead of the Windows backslash ('\'). For students who want to run under Windows, it is their responsibility to use the correct hierarchy separator.

Windows users can run Linux using the VirtualBox tool from Oracle. Details on how to acquire and install this tool can be found in the lab setup guide. The Xilinx Customer Education group offers a preconfigured virtual machine image and includes all the tools necessary to run the labs. Some tools and IP do require licenses, which are the responsibility of the student.

Several environment variables, described in detail in the lab setup guide, which also discusses how you can customize the environment for your situation, are provided for ease of use. Most notably, many environment variables have been defined to reduce the amount of typing you need to do when entering paths.

These environment variables are:

- XILINX_PATH - points to the installation directory for the Xilinx tools. Tools such as the Vivado Design Suite and SDK can be found here.
- PETALINUX_PATH - points to the installation directory for the Xilinx PetaLinux tools. Often this is within XILINX_PATH, but some choose to install it elsewhere.
- TRAINING_PATH - points to the space allocated for students to work through their labs. This directory includes prebuilt images and starting points for the labs and demos.
- usingTools - points to the specific name of the topic within the TRAINING_PATH directory.

  This lab uses $usingTools to indicate the directory for this lab, which is equivalent to $TRAINING_PATH/usingTools.

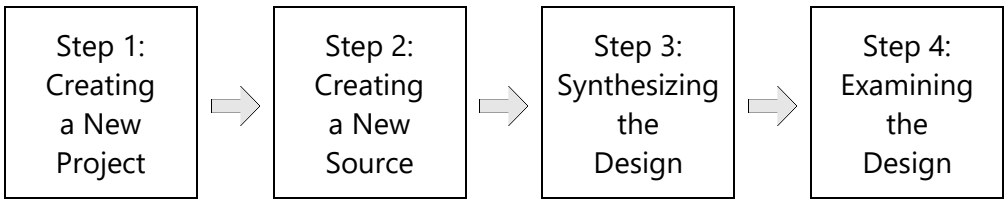
These environment variables work well in the Linux terminal and the Vivado Design Suite and scripts, but not in Eclipse-based tools such as the SDK tool and the SDx™ environment. When using the Vivado tools, you can use $usingTools and the tools will automatically expand the environment variable for you.

© Copyright 2019 Xilinx

However, when you see the notation $usingTools/workspace in SDK or the SDx environment, you should interpret it in its expanded form. The expanded form can be found by opening a terminal window and enter "echo $usingTools".

This lab assumes that you are starting with a Linux environment. If you are working on a Windows system, refer to the lab setup guide to see how to use the Oracle VirtualBox application to run the Xilinx Customer Education pre-configured Ubuntu Linux virtual machine.

General Flow



Creating a New Project Step 1

There are a number of ways to launch the Vivado Design Suite. The two most popular mechanisms are shown here.

1-1. This can be done in two standard ways, use your preferred method.

1-1-1. [Windows 7 users]: Select **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2019.1** > **Vivado 2019.1**.

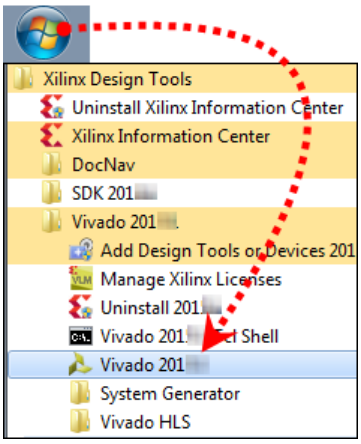


Figure 1-1: Launching the Vivado Design Suite from the Start Menu

[Windows 10 users]: Select **Start** > **Xilinx Design Tools** > **Vivado 2019.1**.

You can double-click the **Vivado Design Suite** shortcut icon ( ) from the Windows or Linux desktop or taskbar.

© Copyright 2019 Xilinx

[Linux users]: If your system has not been configured with a shortcut on the desktop or taskbar, refer to the "Launching the Vivado IDE using a Linux Terminal" section in the *Lab Reference Guide*.

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.

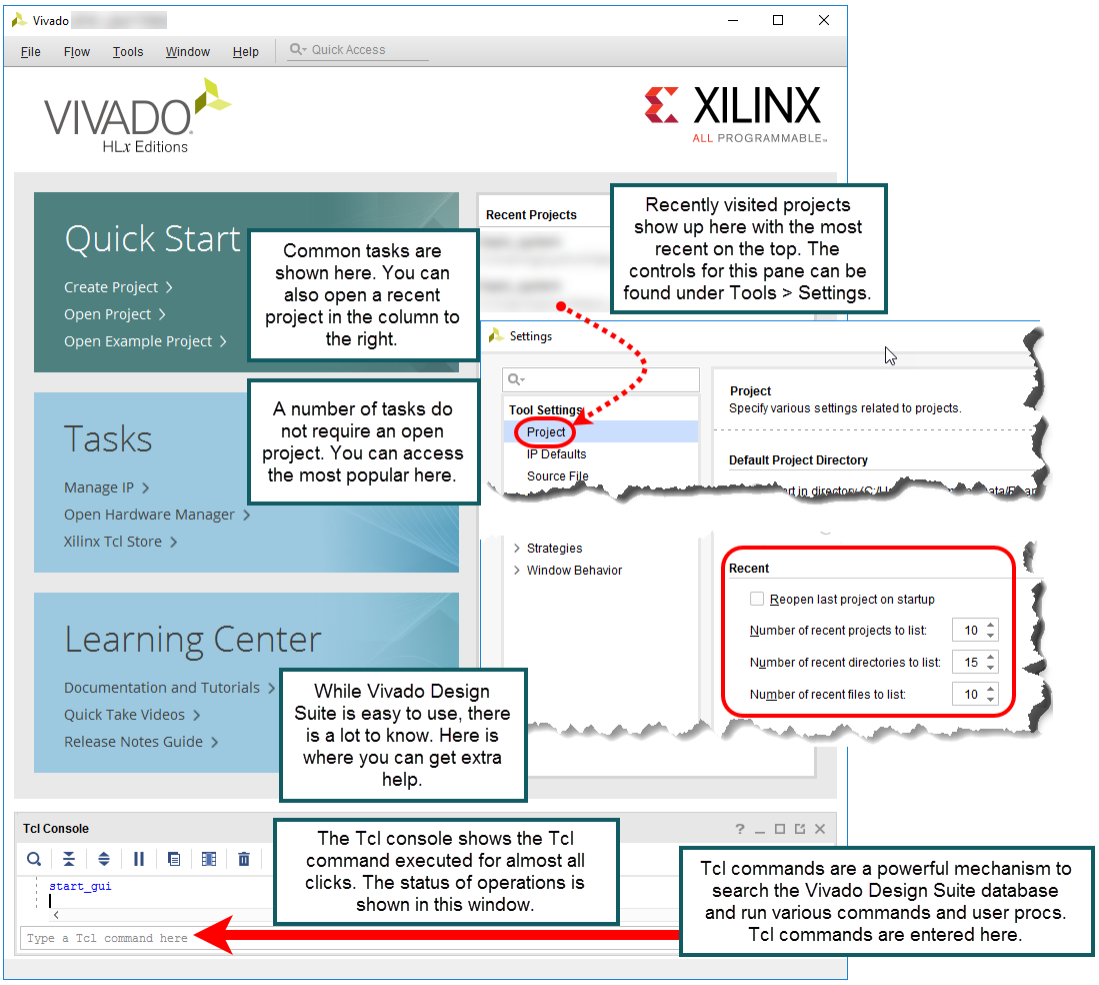


Figure 1-2: Vivado Design Suite Welcome Screen

© Copyright 2019 Xilinx

"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).

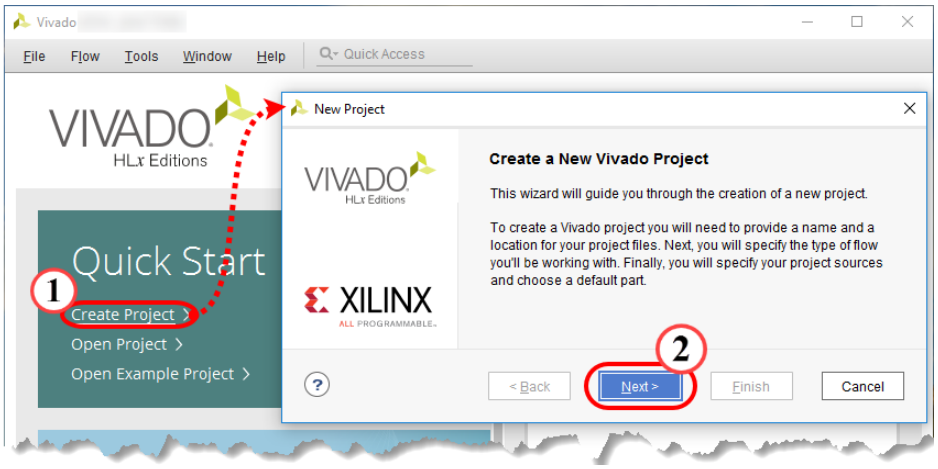


Figure 1-3: Creating a New Vivado Design Suite Project

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).

© Copyright 2019 Xilinx

1-3. You will now encounter a series of dialog boxes asking you to enter different pieces of information describing the project.

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

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

```
$usingTools/lab/KCU105
```

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 as leaving this checked will create an unnecessary level of hierarchy for this lab (3).

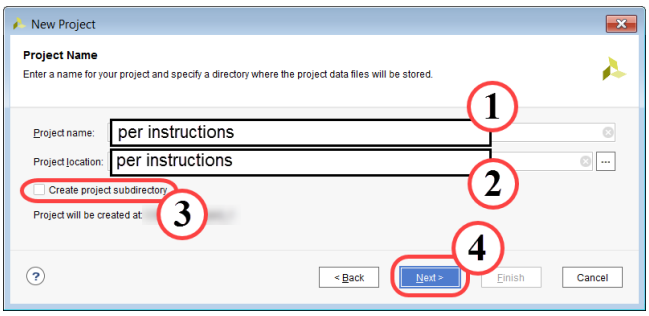


Figure 1-4: Entering the Project Name and Location

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).

1-3-6. Select **Do not specify sources at this time** (2), 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.

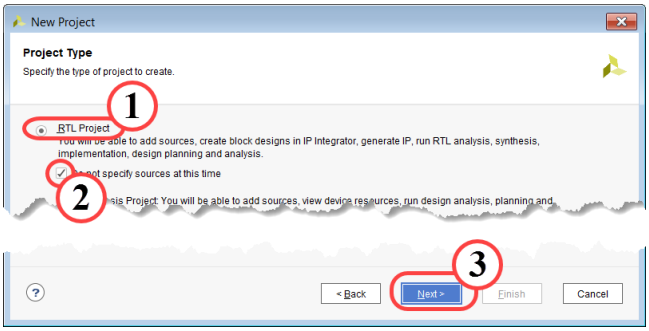


Figure 1-5: Selecting Project Type

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

© Copyright 2019 Xilinx

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. Select **Boards** from the Select 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.

Alternatively you can select the board directly from the list at any time while in this dialog box.

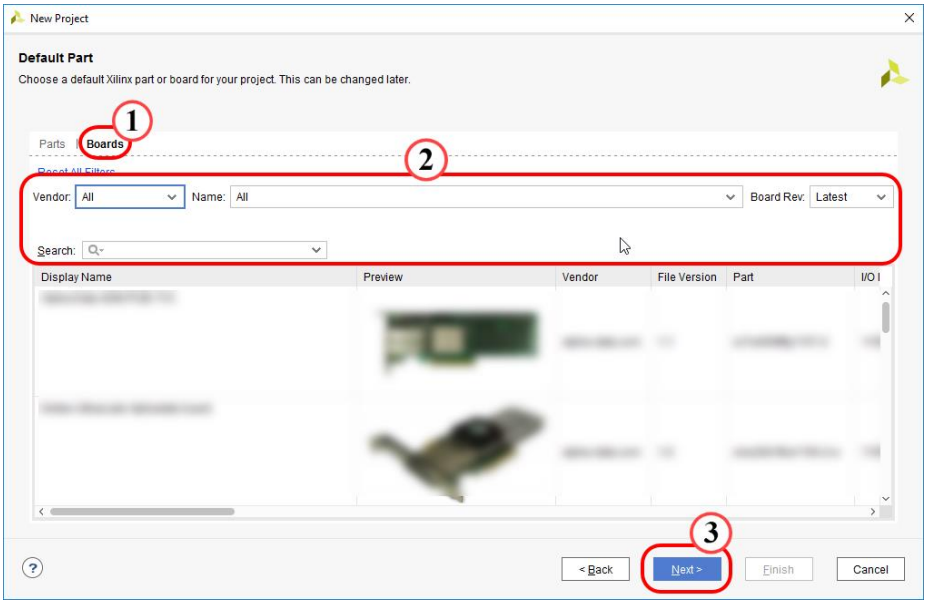


Figure 1-6: Selecting the Board for the Project

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 and making the correction, or you can create the project now and edit the project properties, add or remove files, etc., later.

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.

1-5. Set VHDL as the target language.

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

1-5-2. Select **VHDL** from the Target Language field drop-down menu.

1-5-3. Click **OK** in the Settings dialog box.

© Copyright 2019 Xilinx

Creating a New VHDL Source Step 2

Now that the project has been created, it is time to begin creating the design. The design is a simple 8-bit register with an enable.

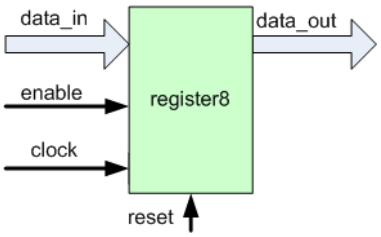


Figure 1-7: register8 Block Diagram

You will begin by creating a VHDL module template, then filling the template with a snippet from the Language Templates and making two simple modifications.

2-1. Create a new HDL source file called *register8*.

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

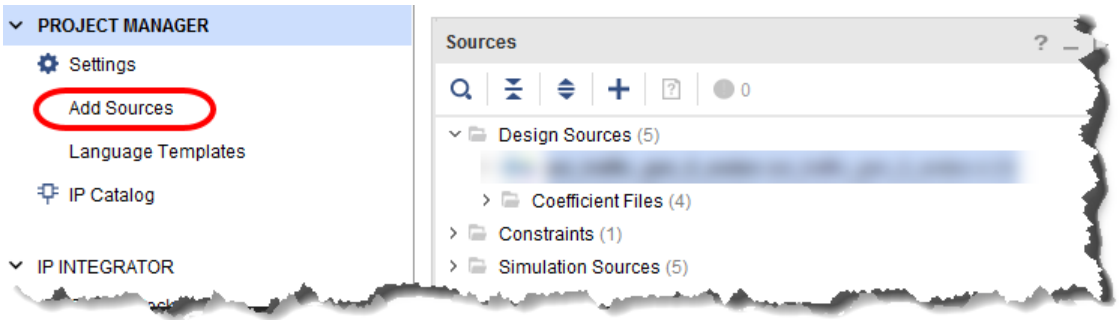


Figure 1-8: Selecting Add Sources

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

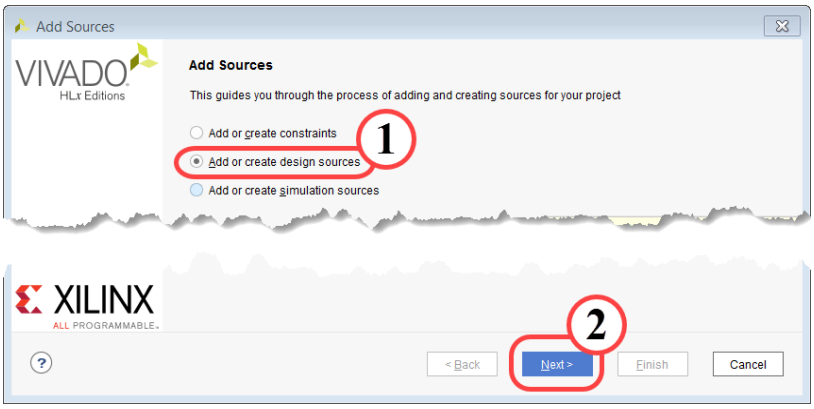


Figure 1-9: Selecting Add or Create Design Sources

© Copyright 2019 Xilinx

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

The Add or Create Design Sources dialog box opens.

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

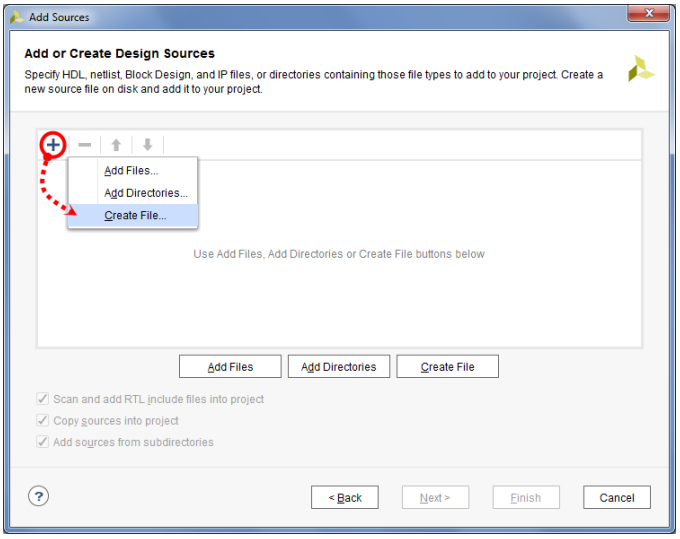


Figure 1-10: Selecting Create File

The Create Source File dialog box opens.

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

2-1-6. Enter **register8** as the file name.

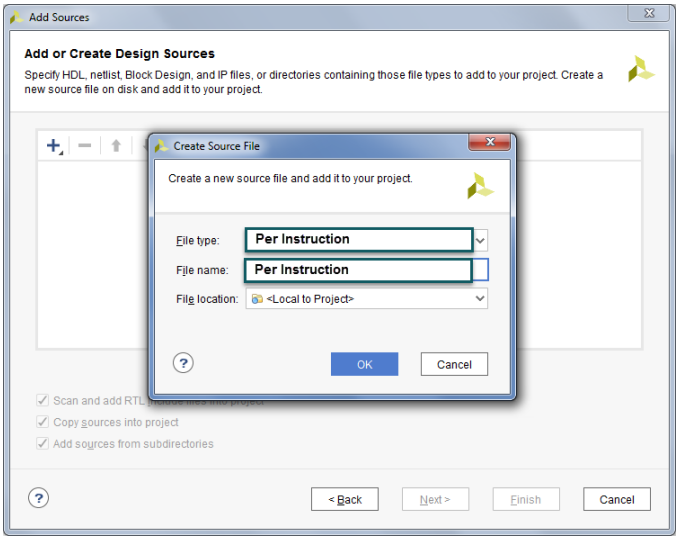


Figure 1-11: Entering File Name and Type

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.

© Copyright 2019 Xilinx

2-2. Specify the inputs and outputs for the *register8.vhd* module in the Define Module dialog box.

- **clock – in – std_logic**
- **reset – in – std_logic**
- **enable – in – std_logic**
- **data_in – in - std_logic_vector – width 8 (i.e., 7 downto 0)**
- **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.

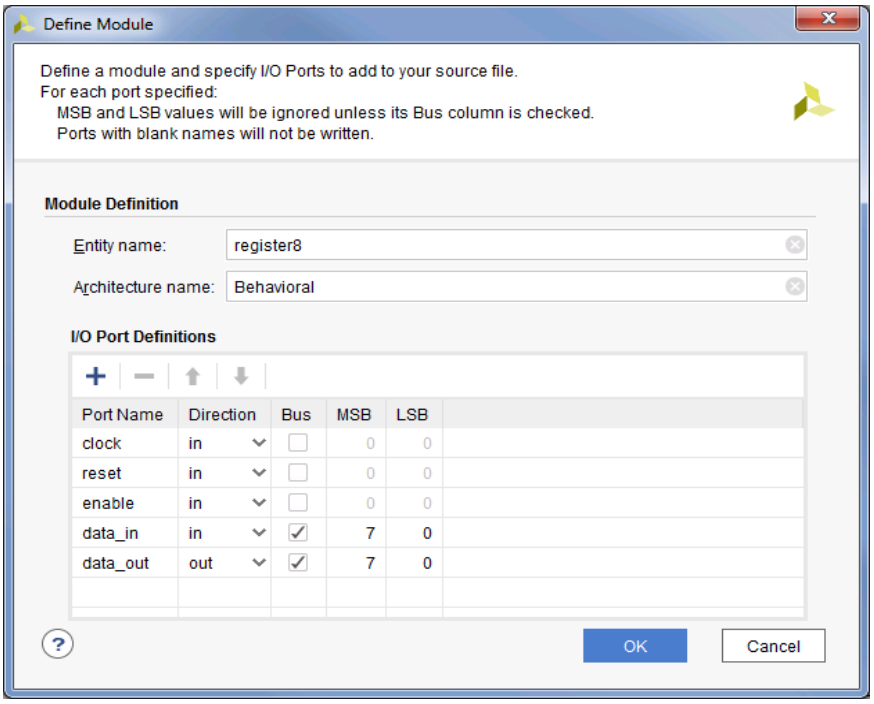


Figure 1-12: Define Module Dialog Box

Hint: Use the + button to add more ports to the module.

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

The newly formed code will appear in the Sources view.

2-2-3. Double-click the **register8.vhd** module in the Hierarchy tab of the Sources window to open the file in a text editor window of the Vivado IDE.

© Copyright 2019 Xilinx

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.

Another package in the IEEE library that is frequently used is the NUMERIC_STD package, which is typically used for managing std_logic_vector and integer conversions. This approved standard package replaces the STD_LOGIC_ARITH, STD_LOGIC_UNSIGNED, and STD_LOGIC_SIGNED packages that you may see in older code.

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 the **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
34  entity register8 is
35      Port ( clock : in STD_LOGIC;
36             reset : in STD_LOGIC;
37             enable : in STD_LOGIC;
38             data_in : in STD_LOGIC_VECTOR (7 downto 0);
39             data_out : out STD_LOGIC_VECTOR (7 downto 0));
40  end register8;
41
42  architecture Behavioral
```

Figure 1-13: Source Code with the NUMERIC_STD Package Commented

© Copyright 2019 Xilinx

Notice how the editor shows that line 27 is no longer a comment by changing from the dull gray color to purple (used for keywords) and black (used for names).

```
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 1-14: Current State of the Source

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

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

2-4. Using the Language Templates, select the snippet for a simple register with enable.

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

2-4-1. Select **Tools** > **Language Templates** to open the Language Templates window.

The Language Templates icon will also be present on the toolbar to the top of the text editor. The Language Templates window contain a large number of code snippets that demonstrate how various constructs can be built.

2-4-2. Expand **VHDL** > **Synthesis Constructs** > **Coding Examples** > **Flip Flops** > **D Flip Flop** > **Posedge** so that you can see the various examples.

© Copyright 2019 Xilinx

2-4-3. Open the **w/Synchronous Active High Reset and CE** example.

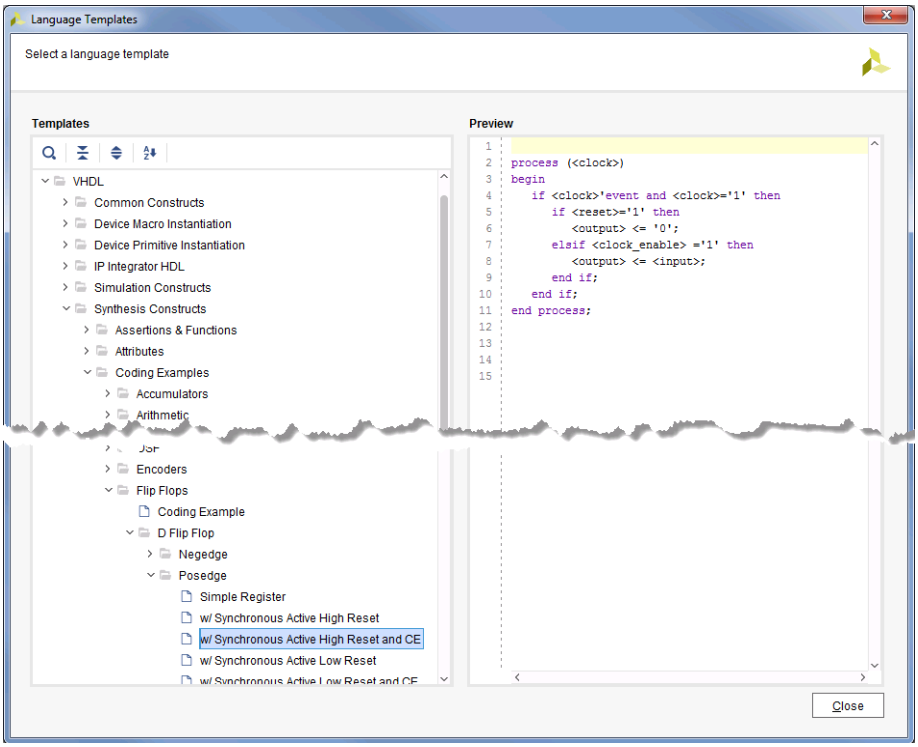


Figure 1-15: Partially Expanded Language Template List

Notice how code appears in the Preview section. This code represents a template for creating the object selected from the tree.

2-5. Copy and paste the entire snippet into the source code between the "begin" and "end Behavioral" statements. Indent the snippet.

2-5-1. Select the text in the Preview section of the Language Templates window.

2-5-2. Right-click the highlighted area to bring up the pop-up menu and select **Copy**.

You can also use the keyboard shortcut <**Ctrl** + **C**>.

2-5-3. Close the Language Templates window.

2-5-4. Click the **register8.vhd** tab to switch back to the source code.

The asterisk means that this file is unsaved.

2-5-5. Click line 45 or 46 (anywhere between the "begin" and "end Behavioral" statements) to mark where you want to place the text that have in the copy buffer.

2-5-6. Right-click and select **Paste**.

You can also use the keyboard shortcut <**Ctrl** + **V**>.

2-5-7. Indent the block of code just inserted by selecting it and pressing the <**Tab**> key.

Good design practice calls for intelligent use of white space.

© Copyright 2019 Xilinx

2-6. The snippet represents generic code. Change the items enclosed in the angle brackets with signals that were defined in the entity.

2-6-1. Make the following substitutions:

- <clock> becomes clock
- <reset> becomes reset
- <clock_enable> becomes enable
- <input> becomes data_in
- <output> becomes data_out

The code should appear as in the following figure.

```
42 architecture Behavioral of register8 is
43
44 begin
45 process (clock)
46 begin
47    if clock'event and clock='1' then
48       if reset='1' then
49          data_out <= '0';
50       elsif enable ='1' then
51          data_out <= data_in;
52       end if;
53    end if;
54 end process;
```

Figure 1-16: Completed Source Code

2-6-2. Observe how the assignment of '0' to data_out is highlighted in the text editor. By hovering the cursor near the line, you will see why the statement is highlighted.

You will fix this issue in the following step.

2-6-3. Select **File** > **Text Editor** > **Save File**.

2-7. Optional: Return to the Language Templates and explore the code snippets for creating specific structures as well as the "Common Constructs" which is a syntax helper.

© Copyright 2019 Xilinx

Synthesizing the Design Step 3

Now that the code has been entered, you must verify that it is what you intended.

3-1. Launch synthesis. Observe how an error is reported in the Console window. This is expected and you will fix this in the next sub-step.

3-1-1. Select **Synthesis** > **Run Synthesis** in the Flow Navigator and click **OK** to launch runs.

A dialog box opens indicating that synthesis has failed.

3-1-2. Click **OK** to close the Synthesis Failed dialog box.

Notice the errors that appear in the Messages console.

3-1-3. Click the **"register8.vhd:<line-number>"** hyperlink, which will open the *register8.vhd* file.

Notice the line number where the error has occurred.

3-2. The error relates to the data_out signal being comprised of eight signals, but the initialization only covers one signal.

Correct the error and resynthesize.

3-2-1. Change the line that reads: "data_out <= '0';" which refers to a single value being assigned to "data_out <= (others=>'0');" which will set the entire bus to zero.

The snippet was valid for non-bus signals. By changing it, you made it compatible with bus signals.

The corrected code should appear as in the figure below.

```
42  architecture Behavioral of register8 is
43
44  begin
45  process (clock)
46  begin
47     if clock'event and clock='1' then
48        if reset='1' then
49           data_out <= (others=>'0');
50        elsif enable ='1' then
51           data_out <= data_in;
52        end if;
53     end if;
54  end process;
55
56  end Behavioral;
```

Figure 1-17: Completed Source Code

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

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

This time, the code should complete without errors.

© Copyright 2019 Xilinx

Examining the Design with Viewers 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 subsequent labs. In this lab, you will view the generated code using the schematic viewers available in the Vivado IDE.

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

The Schematic viewer shows the logic and register relationships and how it is to be implemented.

4-1-1. Select **Open Synthesized Design** in the Synthesis Completed dialog box after synthesis has finished.

4-1-2. Click **OK**.

4-1-3. Select **Layout** > **I/O Planning,** if it is not already selected, to open the I/O Planning view.

Notice that the Vivado IDE view changes; the Package and Device views should now be visible.

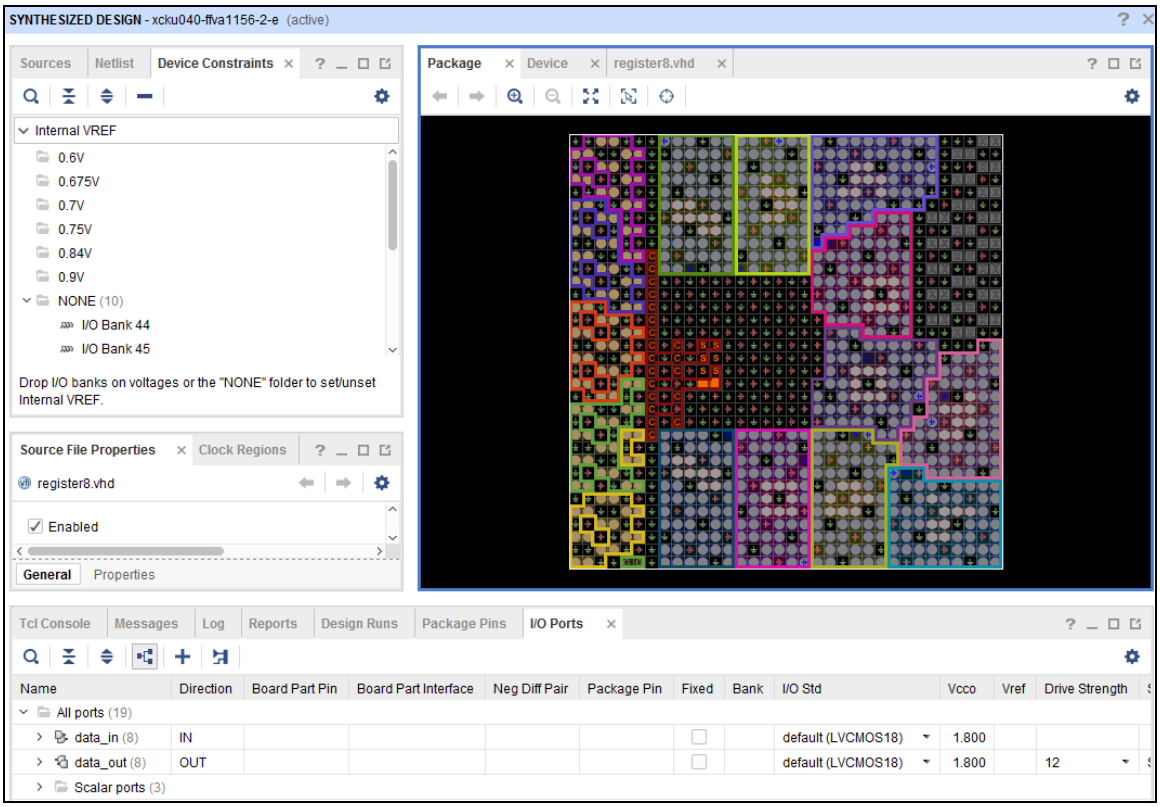


Figure 1-18: Synthesized Design View

4-1-4. Select the **Netlist** tab to the left of Device Constraints tab to see the Netlist view.

© Copyright 2019 Xilinx

4-1-5. Right-click **register8** in the Netlist view and select **Schematic**.

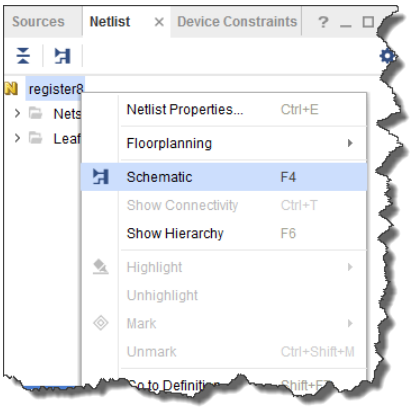


Figure 1-19: Selecting the Schematic of the register8 Netlist

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

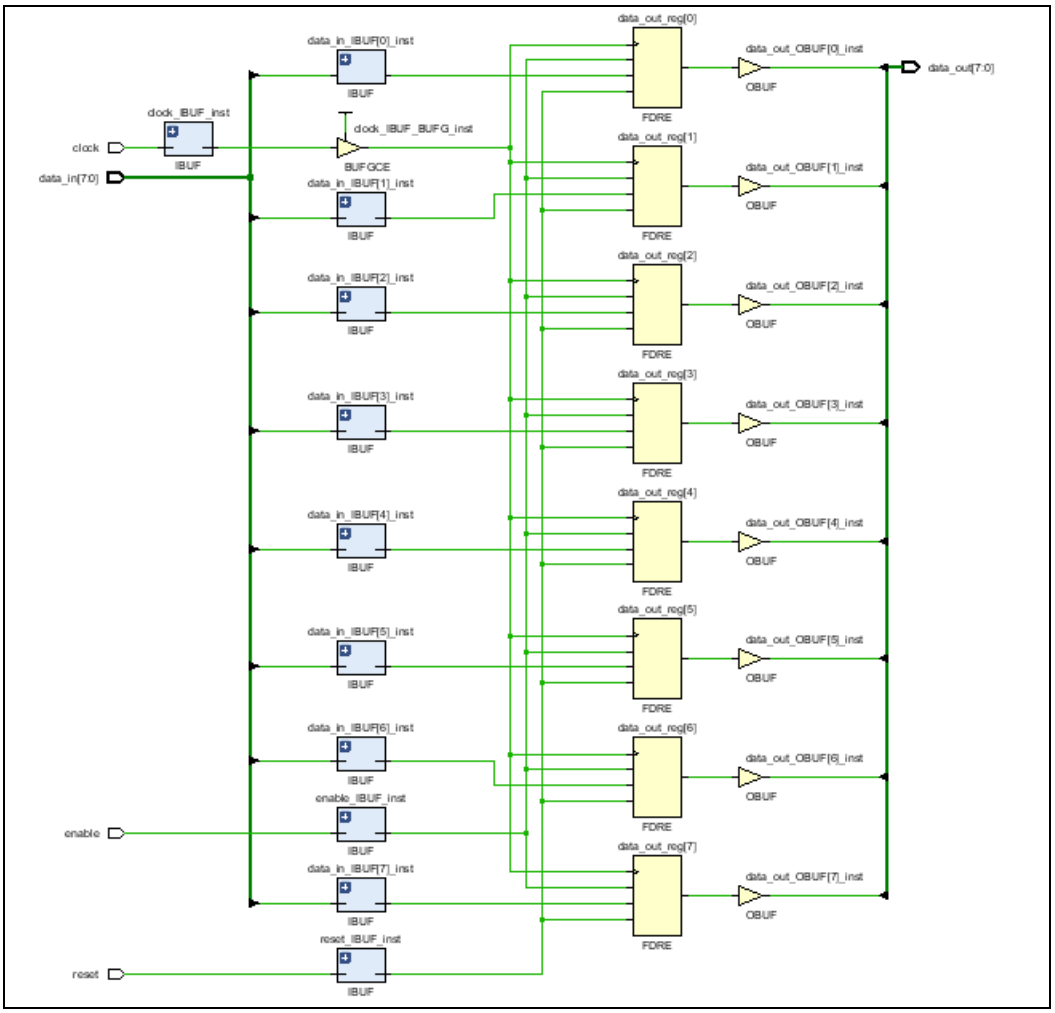


Figure 1-20: Schematic View (KCU105)

© Copyright 2019 Xilinx

Question 1

Does the final design in the Schematic view appear as you would expect? Why or why not?

_____

_____

_____

4-2. Close the Vivado Design Suite.

4-2-1. Select **File** > **Exit**.

The Exit Vivado dialog box opens.

Figure 1-21: Exit Vivado Dialog Box

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

[Optional]: Clean up the file system.

Some systems 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. Both processes will recursively delete all of the files in the `$usingTools` directory.

4-3. Clean up the VirtualBox file system via the GUI.

4-3-1. Using the graphical browser (Windows: press the <**Windows**> key + <**E**>; Linux: press <**Ctrl** + **N**>), navigate to `$usingTools`.

4-3-2. Select **`usingTools`**.

4-3-3. Press <**Delete**>.

© Copyright 2019 Xilinx

4-4. Clean up the VirtualBox file system via the command line.

4-4-1. Open a terminal window (Windows: press the <**Windows**> key + <**R**>, then enter **cmd**; Linux: press <**Ctrl** + **Alt** + **T**>)

4-4-2. Enter the following command to delete the contents of the workspace:

[Windows users]: `rd /s /q $usingTools`

[Linux users]: `rm -rf $usingTools`

Summary

This lab demonstrated the basic capabilities of the Vivado Design Suite, covering design entry, compilation, and resultant schematic viewing.

Additional features and capabilities will be shown in subsequent labs.

© Copyright 2019 Xilinx

Answers

1. Does the final design in the Schematic view appear as you would expect? Why or why not?

   It should be pretty close to what you expected. Eight registers, each with an enable, clock, data in, data out, and a reset were built. The tools have also inserted the buffers for the Input and Output signals and have used some clock buffers for the clock pins.

© Copyright 2019 Xilinx

Lab 2: Using Concurrent Statements

2019.1

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.

© Copyright 2019 Xilinx

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.

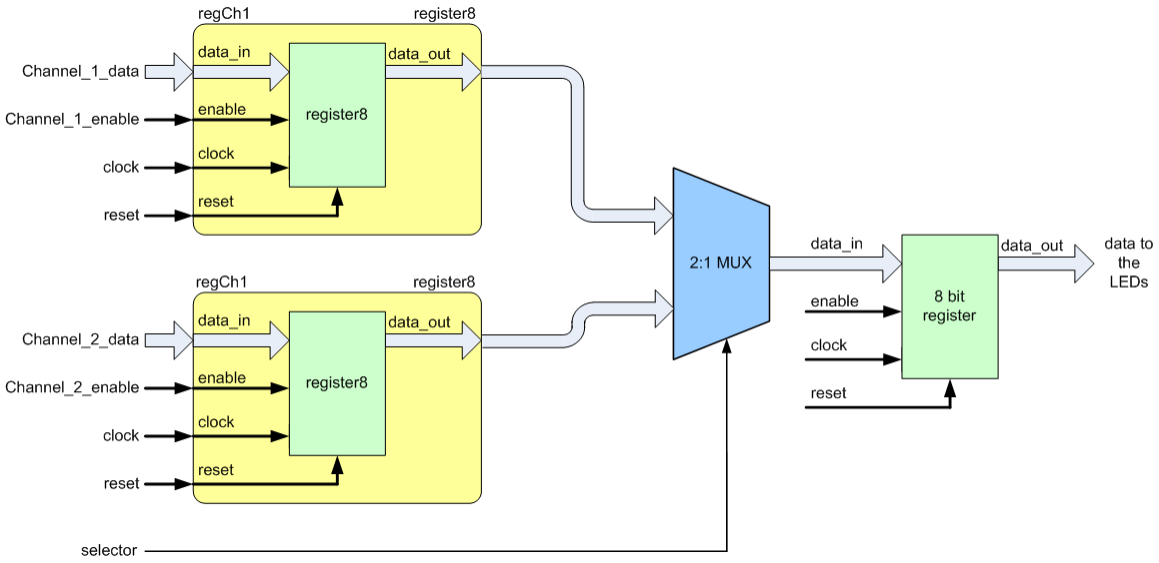


Figure 2-1: Block Diagram for This Lab

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 under the provided Linux platform. The provided Linux platform has a customized lab environment. These customizations have been made to simplify and enhance your learning experience. Many labs and demos can be successfully executed in the Windows environment as well. Topics including PetaLinux, Yocto, and QEMU are generally not supported under Windows.

© Copyright 2019 Xilinx

The instructions, for the sake of brevity, are expressed using the Linux notation. This includes the forward slash ('/') as the hierarchy separator instead of the Windows backslash ('\'). For students who want to run under Windows, it is their responsibility to use the correct hierarchy separator.

Windows users can run Linux using the VirtualBox tool from Oracle. Details on how to acquire and install this tool can be found in the lab setup guide. The Xilinx Customer Education group offers a preconfigured virtual machine image and includes all the tools necessary to run the labs. Some tools and IP do require licenses, which are the responsibility of the student.

Several environment variables, described in detail in the lab setup guide, which also discusses how you can customize the environment for your situation, are provided for ease of use. Most notably, many environment variables have been defined to reduce the amount of typing you need to do when entering paths.

These environment variables are:

- XILINX_PATH - points to the installation directory for the Xilinx tools. Tools such as the Vivado Design Suite and SDK can be found here.
- PETALINUX_PATH - points to the installation directory for the Xilinx PetaLinux tools. Often this is within XILINX_PATH, but some choose to install it elsewhere.
- TRAINING_PATH - points to the space allocated for students to work through their labs. This directory includes prebuilt images and starting points for the labs and demos.
- concurrrency - points to the specific name of the topic within the TRAINING_PATH directory.

  This lab uses $concurrrency to indicate the directory for this lab, which is equivalent to $TRAINING_PATH/concurrrency.

These environment variables work well in the Linux terminal and the Vivado Design Suite and scripts, but not in Eclipse-based tools such as the SDK tool and the SDx™ environment. When using the Vivado tools, you can use $concurrrency and the tools will automatically expand the environment variable for you.

However, when you see the notation $concurrrency/workspace in SDK or the SDx environment, you should interpret it in its expanded form. The expanded form can be found by opening a terminal window and enter "echo $concurrrency".

This lab assumes that you are starting with a Linux environment. If you are working on a Windows system, refer to the lab setup guide to see how to use the Oracle VirtualBox application to run the Xilinx Customer Education pre-configured Ubuntu Linux virtual machine.

General Flow

© Copyright 2019 Xilinx

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.

There are a number of ways to launch the Vivado Design Suite. The two most popular mechanisms are shown here.

1-1. This can be done in two standard ways, use your preferred method.

1-1-1. [Windows 7 users]: Select **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2019.1** > **Vivado 2019.1**.

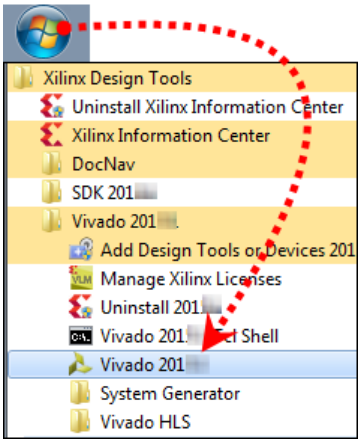


Figure 2-2: Launching the Vivado Design Suite from the Start Menu

[Windows 10 users]: Select **Start** > **Xilinx Design Tools** > **Vivado 2019.1**.

You can double-click the **Vivado Design Suite** shortcut icon ( ) from the Windows or Linux desktop or taskbar.

[Linux users]: If your system has not been configured with a shortcut on the desktop or taskbar, refer to the "Launching the Vivado IDE using a Linux Terminal" section in the *Lab Reference Guide*.

© Copyright 2019 Xilinx

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.

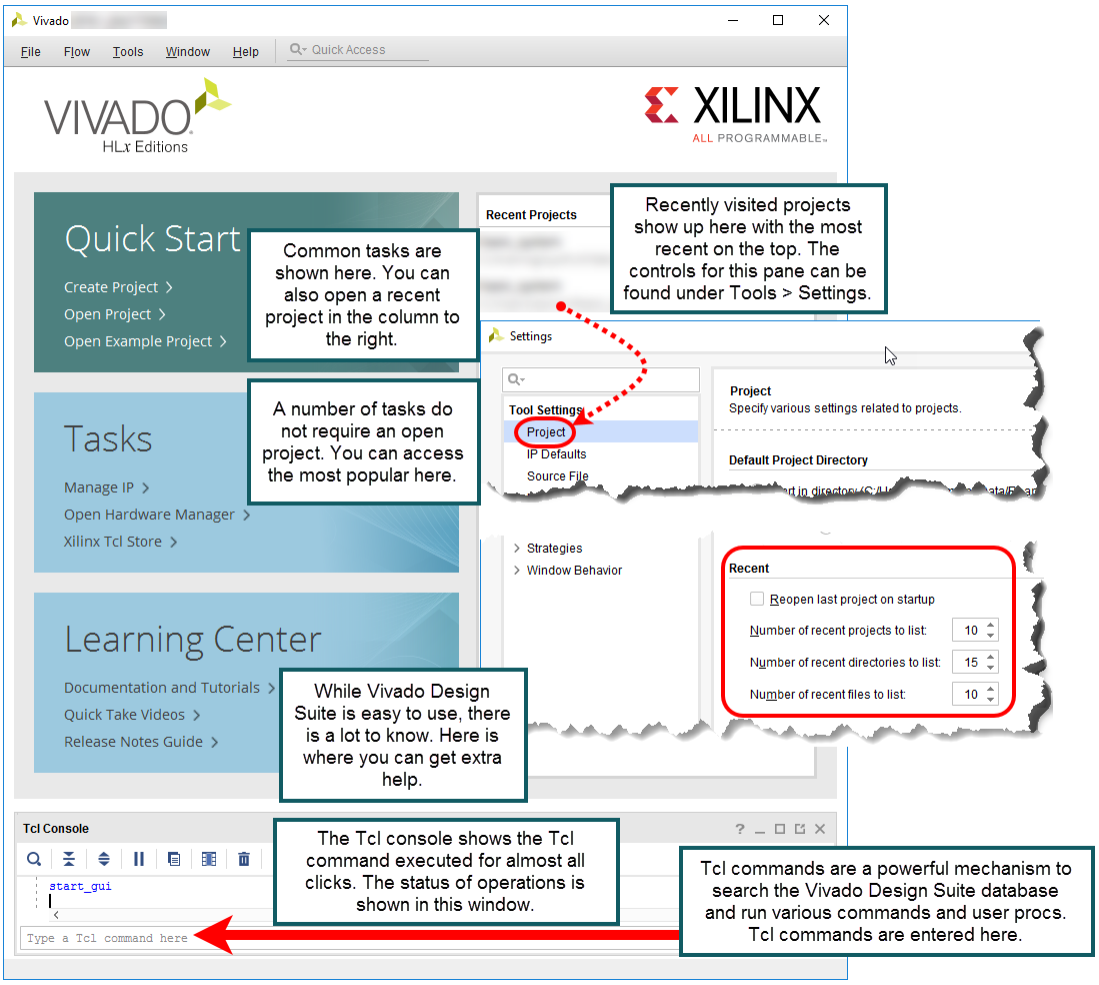


Figure 2-3: Vivado Design Suite Welcome Screen

© Copyright 2019 Xilinx

"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).

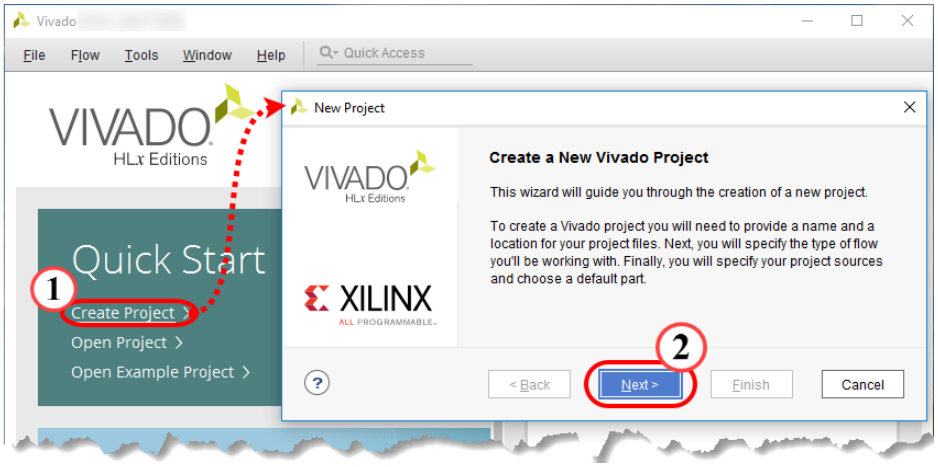


Figure 2-4: Creating a New Vivado Design Suite Project

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).

© Copyright 2019 Xilinx

1-3. You will now encounter a series of dialog boxes asking you to enter different pieces of information describing 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):

```
$concurrency/lab/KCU105
```

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 as leaving this checked will create an unnecessary level of hierarchy for this lab (3).

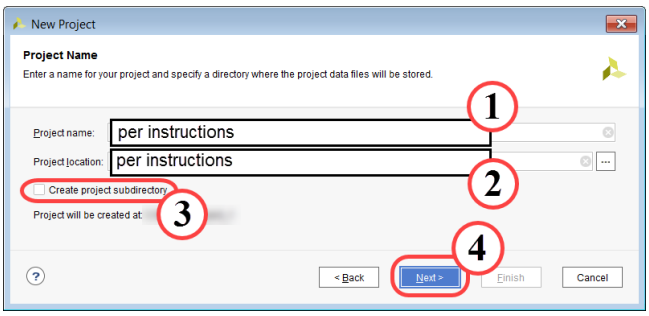


Figure 2-5: Entering the Project Name and Location

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).

1-3-6. Select **Do not specify sources at this time** (2), 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.

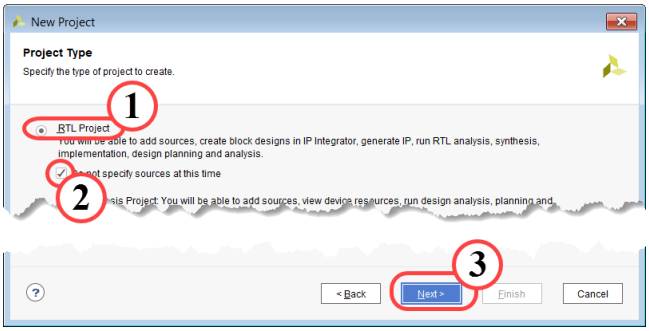


Figure 2-6: Selecting Project Type

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

© Copyright 2019 Xilinx

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. Select **Boards** from the Select 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.

Alternatively you can select the board directly from the list at any time while in this dialog box.

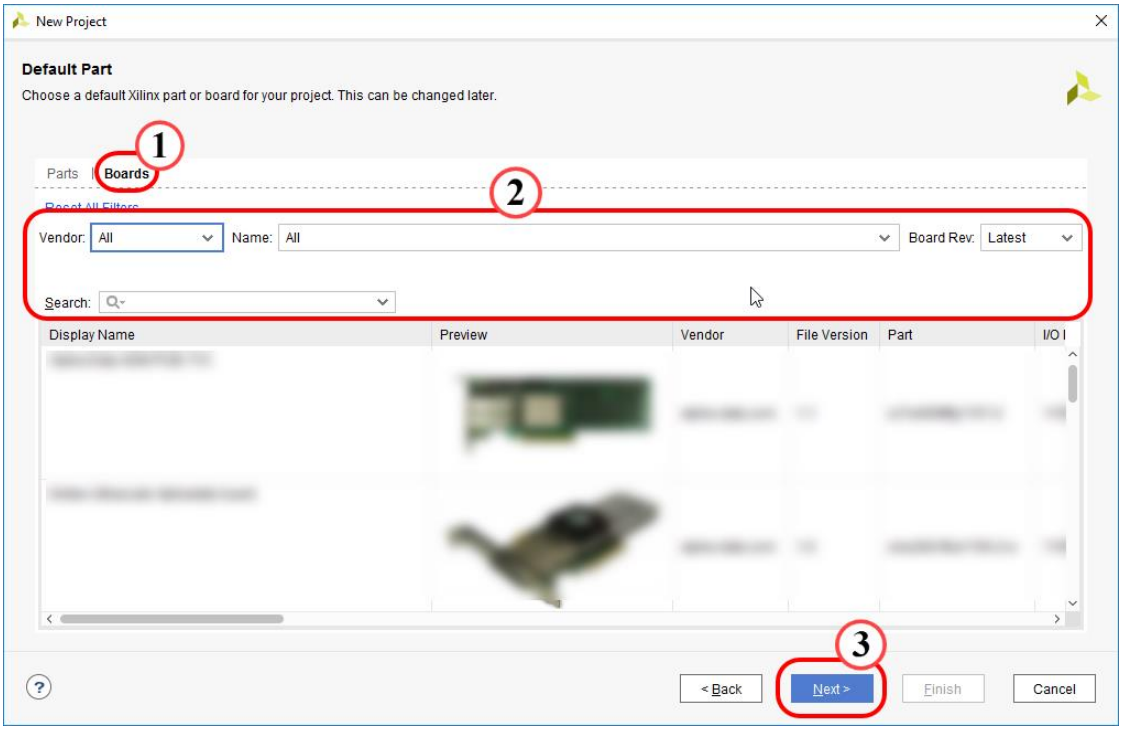


Figure 2-7: Selecting the Board for the Project

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 and making the correction, or you can create the project now and edit the project properties, add or remove files, etc., later.

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**.

© Copyright 2019 Xilinx

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.

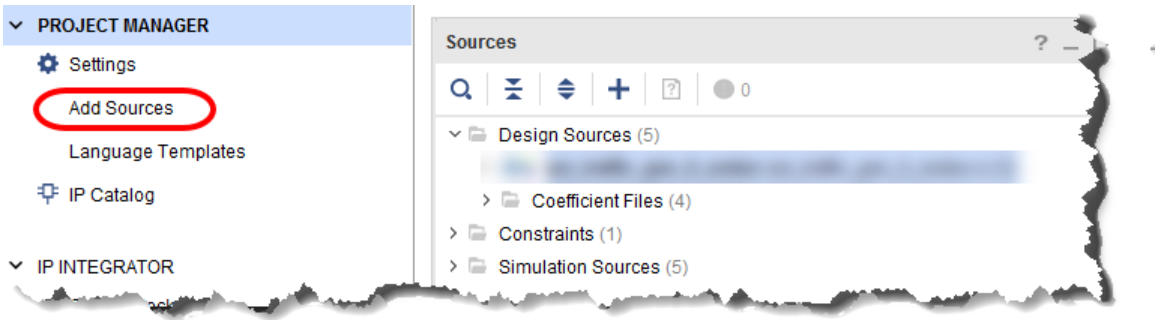


Figure 2-8: Selecting Add Sources

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

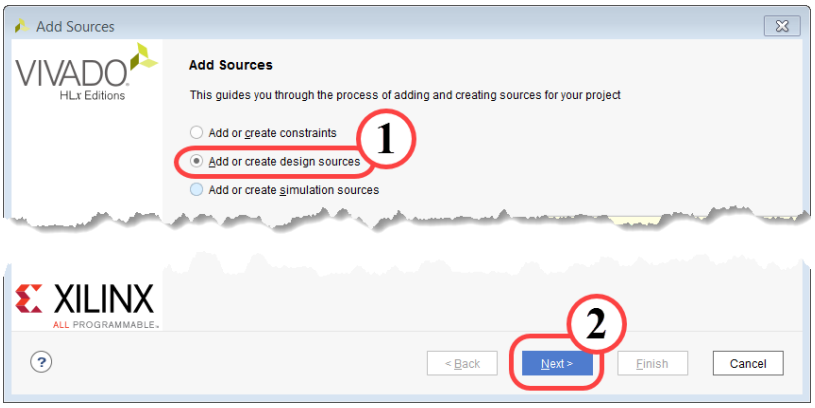


Figure 2-9: Selecting Add or Create Design Sources

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

The Add or Create Design Sources dialog box opens.

© Copyright 2019 Xilinx

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

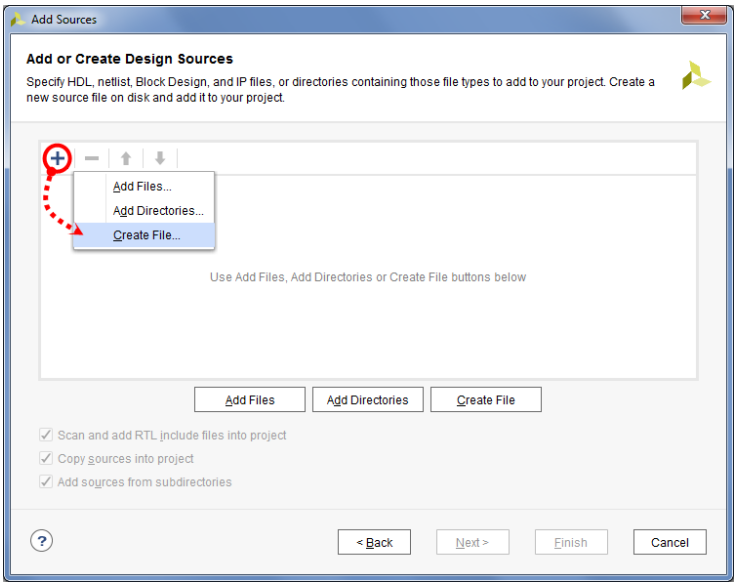


Figure 2-10: Selecting Create File

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.

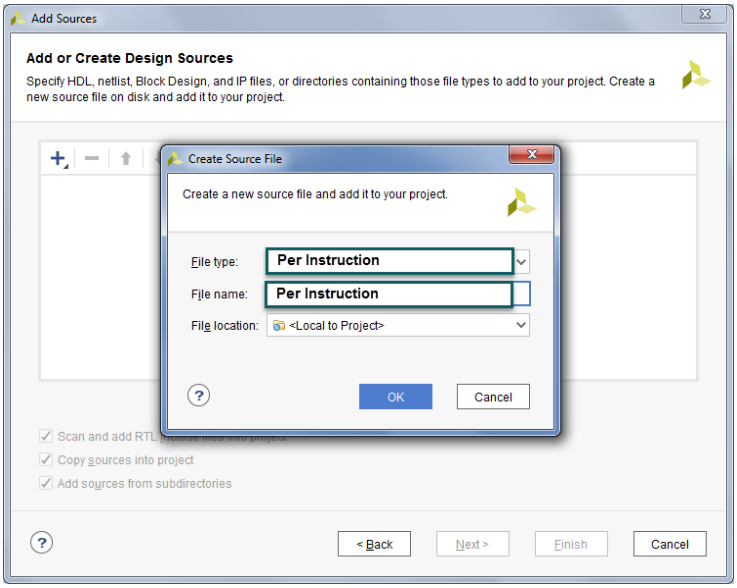


Figure 2-11: Entering File Name and Type

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.

© Copyright 2019 Xilinx

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.

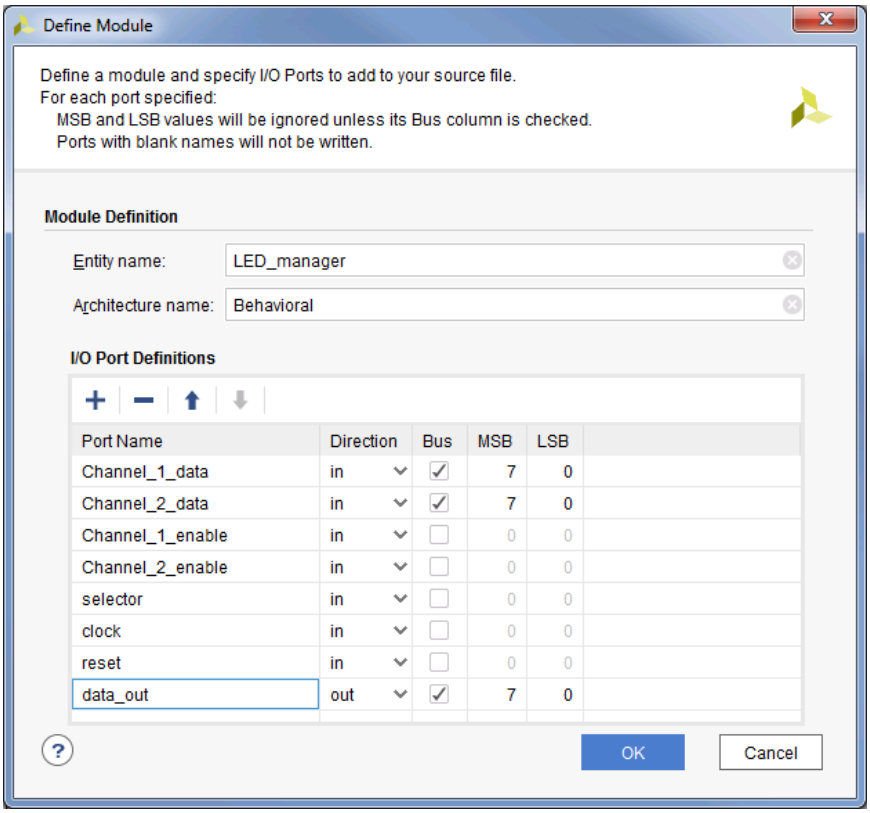


Figure 2-12: Defining Ports for LED_Manager

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.

© Copyright 2019 Xilinx

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 2-13: 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 2-14: 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.**

© Copyright 2019 Xilinx

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**.

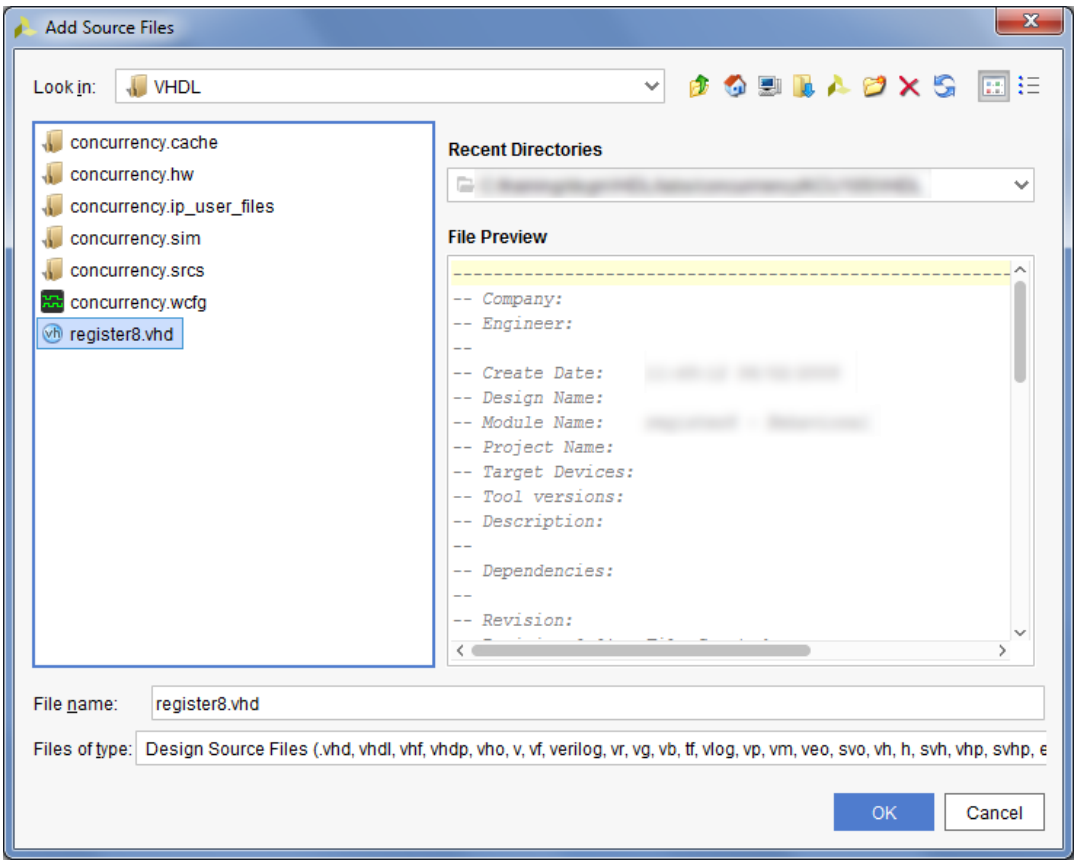


Figure 2-15: Adding register8.vhd to the Project

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.

© Copyright 2019 Xilinx

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 2-16: 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.

© Copyright 2019 Xilinx

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.

```
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 2-17: Two Instantiations of register8 with Inputs Assigned

© Copyright 2019 Xilinx

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_vectors 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 2-18: 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 2-19: Registered Outputs for the reigster8 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.

© Copyright 2019 Xilinx

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?

_____

_____

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?

_____

_____

_____

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.

© Copyright 2019 Xilinx

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 2-20: 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?

© Copyright 2019 Xilinx

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_hierarachy** option to **none** as shown below.

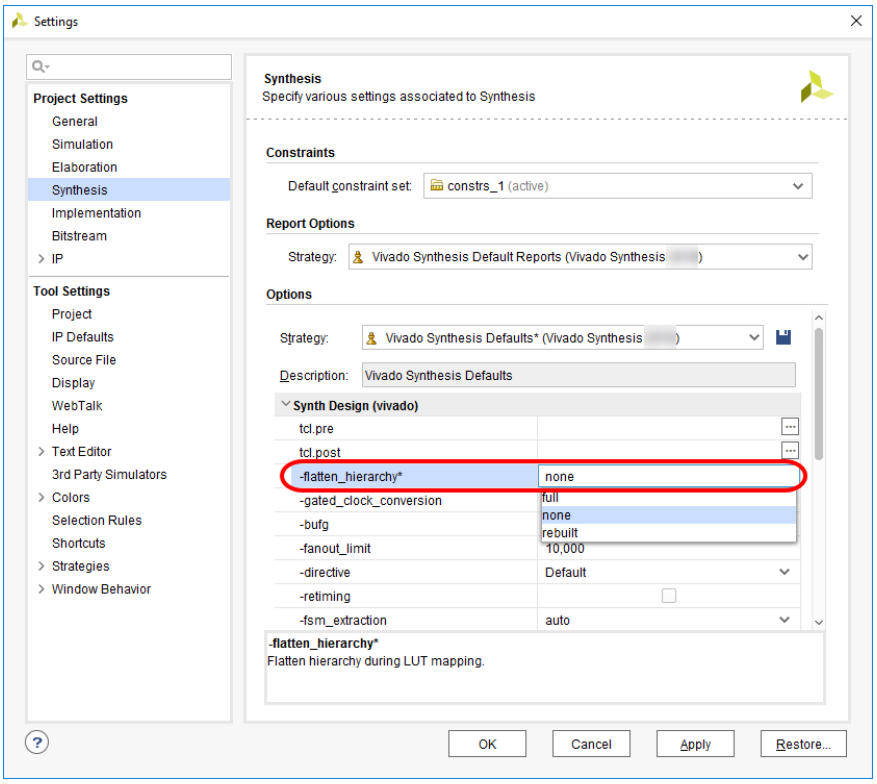


Figure 2-21: Keeping the Hierarchy

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

© Copyright 2019 Xilinx

3-2. Run synthesis.

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

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

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

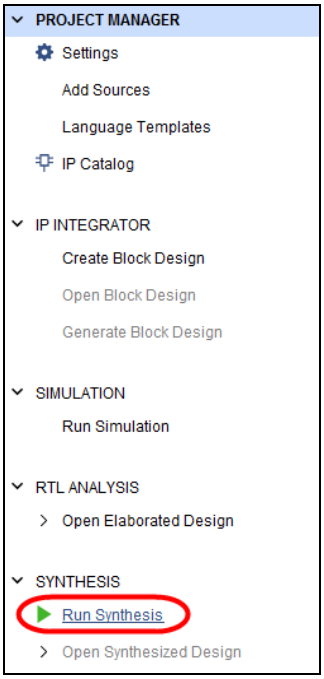


Figure 2-22: Selecting Run Synthesis

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

After the synthesis process completes, the Synthesis Completed dialog box opens. The dialog box prompts you to run implementation, open the synthesized design, or view reports.

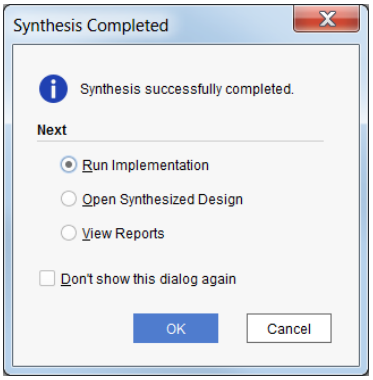


Figure 2-23: Synthesis Completed Dialog Box

3-2-4. Click **OK** to continue with your preferred choice or **Cancel** to simply close the dialog box and return to the normal view of the Vivado Design Suite.

© Copyright 2019 Xilinx

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.

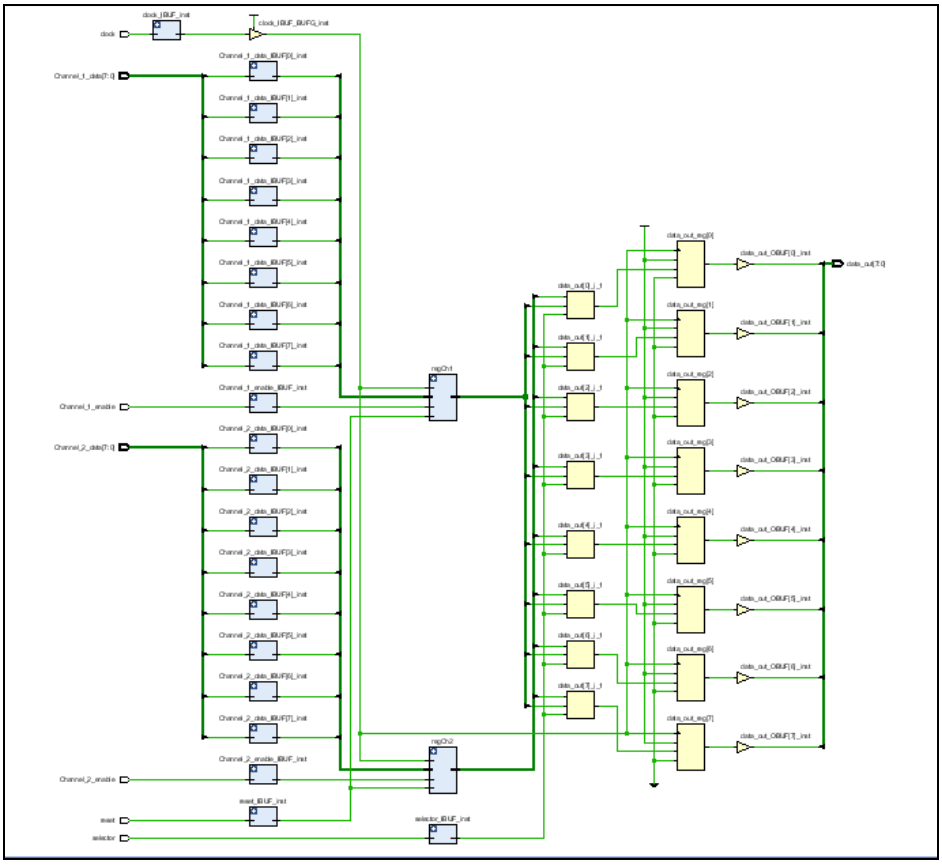


Figure 2-24: Full Schematic View (KCU105)

© Copyright 2019 Xilinx

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.

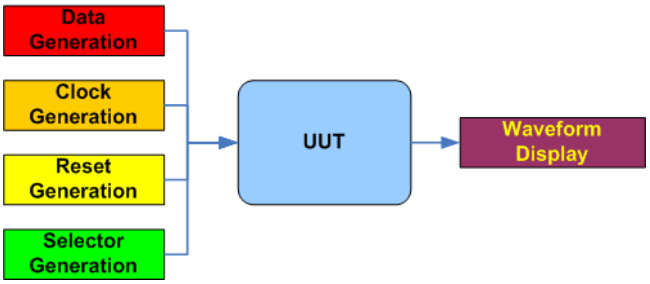


Figure 2-25: Simple Testbench

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.

5-1-2. Select **Add or create simulation sources**.

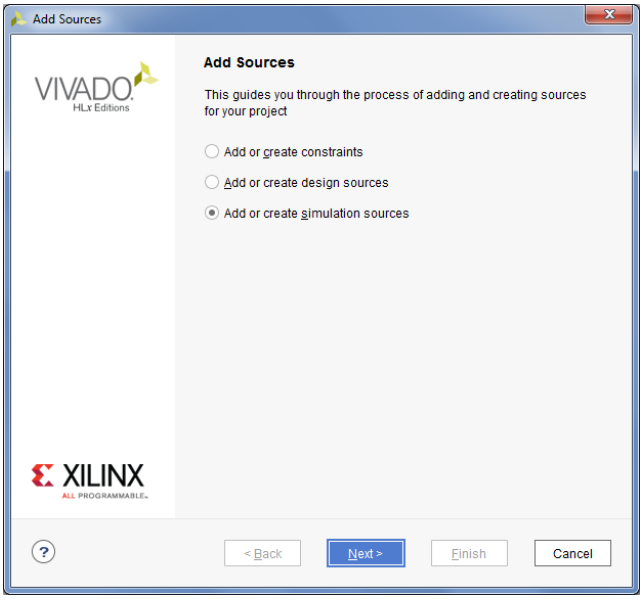


Figure 2-26: Add Sources for Simulation

The Add or Create Simulation Sources dialog box opens.

© Copyright 2019 Xilinx

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?

_____

_____

_____

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)

© Copyright 2019 Xilinx

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;
```

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;
```

Figure 2-29: Component Declaration of the UUT

© Copyright 2019 Xilinx

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

© Copyright 2019 Xilinx

5-6-3. Save the file.

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

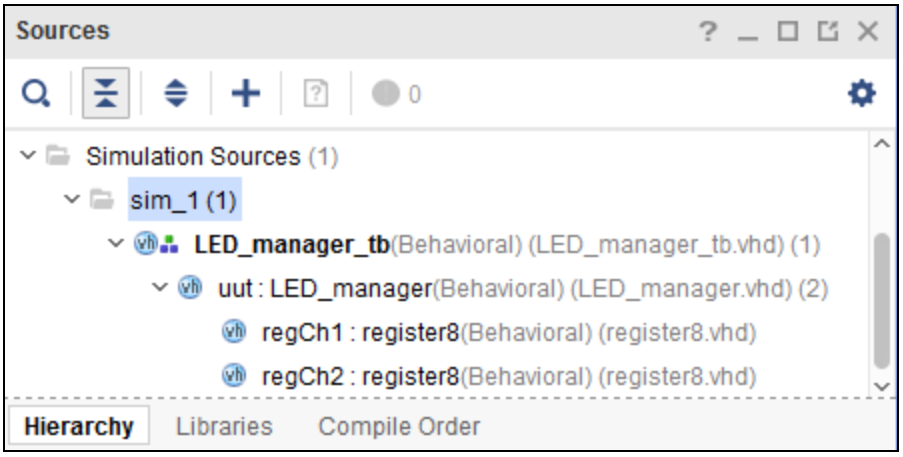


Figure 2-32: UUT Under the Hierarchy of testbench LED_manager_tb

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;
```

Figure 2-34: Generating Reset and Clock Stimulus

© Copyright 2019 Xilinx

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**.

© Copyright 2019 Xilinx

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

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 `$concurrency/lab/KCU105` directory.

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

5-12. Run the simulation for 3 µs.

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?

_____

_____

5-13. Close the simulation.

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

© Copyright 2019 Xilinx

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**.

[Optional]: Clean up the file system.

Some systems 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. Both processes will recursively delete all of the files in the `$concurrrency` directory.

5-15. Clean up the VirtualBox file system via the GUI.

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

5-15-2. Select **`concurrrency`**.

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

5-16. Clean up the VirtualBox file system via the command line.

5-16-1. Open a terminal window (Windows: press the <**Windows**> key + <**R**>, then enter **cmd**; Linux: press <**Ctrl** + **Alt** + **T**>)

5-16-2. Enter the following command to delete the contents of the workspace:

[Windows users]: **`rd /s /q $concurrrency`**

[Linux users]: **`rm -rf $concurrrency`**

© Copyright 2019 Xilinx

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).

© Copyright 2019 Xilinx

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.

© Copyright 2019 Xilinx

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.

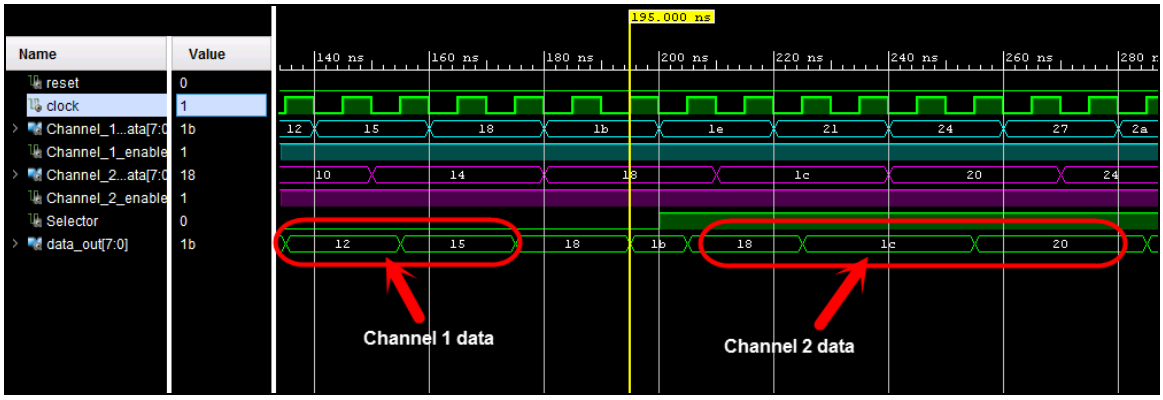


Figure 2-39: Waveform View Zoomed in to See the Values

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.

© Copyright 2019 Xilinx

Lab 3: Designing a Simple Process

2019.1

Abstract

This lab introduces you to the VHDL process. It should take approximately 60 minutes.

Objectives

After completing this lab, you will be able to:

- Create a simple process for managing resets
- Assert reset asynchronously
- Deassert reset synchronously

Introduction

This lab will show you how and when to use the VHDL process. Good design practice calls for the reset condition to be tested synchronously to flip-flops, indicating that processes should contain the reset condition test inside the clock edge detection test.

Sometimes it is best to apply a reset asynchronously (thus ensuring that all logic associated with this signal enter reset at the earliest possible time) and deassert reset synchronously (so that all the logic exits the reset condition at the same time). In order to correctly design this process, you must understand the nature of meta-stability.

Digital logic is built on transistors. Transistors are analog devices, so the output of a transistor is not always a logic ‘0’ or logic ‘1’—there is a transition time. Logic gates are not only described by their behavior, but also by their performance, which includes this transition time.

When dealing with meta-stability, the critical timing parameter is the "set-up and hold" time. "Set-up" indicates that the data must be present for a certain amount of time prior to the arrival of the clock edge and "hold" indicates how long the signal must remain there to be properly acknowledged.

© Copyright 2019 Xilinx

If a signal is applied to a flip-flop nearly simultaneously with a clock edge, the output of the flip-flop is not defined; that is, it cannot be determined if it is a stable ‘0’ or ‘1’. Physically, it is a voltage somewhere between the legal ‘0’ and legal ‘1’ values.

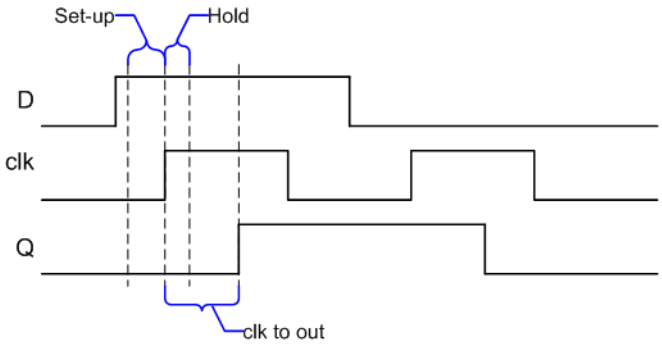


Figure 3-1: Valid Flip-Flop Timing

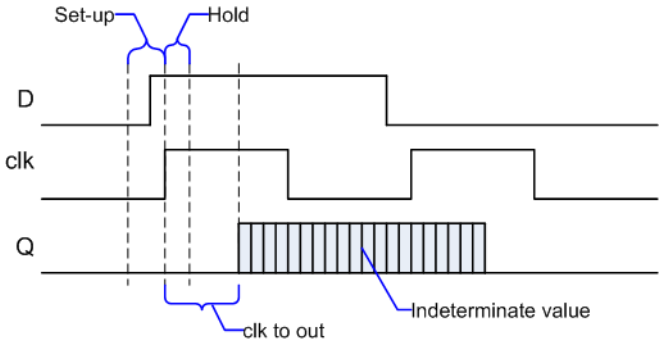


Figure 3-2: Illegal (Bad) Setup Time and Its Consequences

The time that it takes to recover from a meta-stable condition (that is, resolve to a valid value) is dependent on the technology used to produce the underlying transistors. Usually it is sufficient to provide the condition a clock cycle to recover. This is done with a slight modification to the classic meta-stability hardening circuit shown in the figure below.

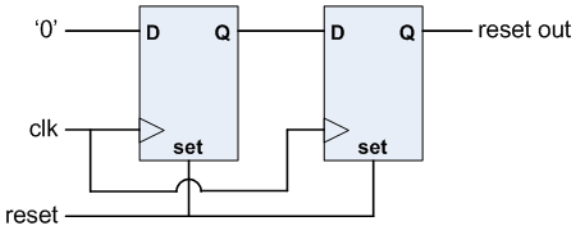


Figure 3-3: Circuit for This Lab

With this circuit, the advent of an active-high reset drives both flip-flops into a Q=’1’ state and the reset out signal is distributed to all the logic connected to it. While it is possible that the synchronous flip-flops receiving this signal may enter a meta-stable state, this will be cleanly resolved by the next clock pulse. In general, it is not as important to enter the reset condition as it is to cleanly exit it.

© Copyright 2019 Xilinx

When the reset is released, the first flip-flop clocks in at logic '0'. Again, this may be meta-stable, but this is where the second flip-flop comes in. The output of the second flip-flop is still stable at '1'. If the first flip-flop becomes meta-stable, it has one clock cycle to recover before being sampled by the second flip-flop.

Finally, on the second clock, the reset going low has been synchronized with the clock.

Note for advanced users: There should be two constraints applied to this circuit. To properly constrain a reset bridge, use the ASYNC_REG property on the flip-flops to have them placed in the same CLB and add a set_max_delay constraint on the async reset input.

Because the lab does not require a specific board for you to download the bitstream, you can choose your own evaluation development board or device (or as specified by your instructor). The figures shown in the lab or the RTL view of your design may vary slightly depending on the device that you have selected.

Understanding the Lab Environment

The labs and demos provided in this course are designed to run under the provided Linux platform. The provided Linux platform has a customized lab environment. These customizations have been made to simplify and enhance your learning experience. Many labs and demos can be successfully executed in the Windows environment as well. Topics including PetaLinux, Yocto, and QEMU are generally not supported under Windows.

The instructions, for the sake of brevity, are expressed using the Linux notation. This includes the forward slash ('/') as the hierarchy separator instead of the Windows backslash ('\'). For students who want to run under Windows, it is their responsibility to use the correct hierarchy separator.

Windows users can run Linux using the VirtualBox tool from Oracle. Details on how to acquire and install this tool can be found in the lab setup guide. The Xilinx Customer Education group offers a preconfigured virtual machine image and includes all the tools necessary to run the labs. Some tools and IP do require licenses, which are the responsibility of the student.

Several environment variables, described in detail in the lab setup guide, which also discusses how you can customize the environment for your situation, are provided for ease of use. Most notably, many environment variables have been defined to reduce the amount of typing you need to do when entering paths.

These environment variables are:

- XILINX_PATH - points to the installation directory for the Xilinx tools. Tools such as the Vivado Design Suite and SDK can be found here.
- PETALINUX_PATH - points to the installation directory for the Xilinx PetaLinux tools. Often this is within XILINX_PATH, but some choose to install it elsewhere.
- TRAINING_PATH - points to the space allocated for students to work through their labs. This directory includes prebuilt images and starting points for the labs and demos.

© Copyright 2019 Xilinx

- processes - points to the specific name of the topic within the TRAINING_PATH directory. This lab uses $processes to indicate the directory for this lab, which is equivalent to $TRAINING_PATH/processes.

These environment variables work well in the Linux terminal and the Vivado Design Suite and scripts, but not in Eclipse-based tools such as the SDK tool and the SDx™ environment. When using the Vivado tools, you can use $processes and the tools will automatically expand the environment variable for you.

However, when you see the notation $processes/workspace in SDK or the SDx environment, you should interpret it in its expanded form. The expanded form can be found by opening a terminal window and enter "echo $processes".

This lab assumes that you are starting with a Linux environment. If you are working on a Windows system, refer to the lab setup guide to see how to use the Oracle VirtualBox application to run the Xilinx Customer Education pre-configured Ubuntu Linux virtual machine.

General Flow

© Copyright 2019 Xilinx

Creating a New Project Step 1

There are a number of ways to launch the Vivado Design Suite. The two most popular mechanisms are shown here.

1-1. This can be done in two standard ways, use your preferred method.

1-1-1. [Windows 7 users]: Select **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2019.1** > **Vivado 2019.1**.

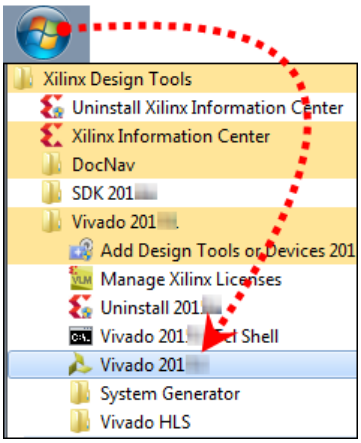


Figure 3-4: Launching the Vivado Design Suite from the Start Menu

[Windows 10 users]: Select **Start** > **Xilinx Design Tools** > **Vivado 2019.1**.

You can double-click the **Vivado Design Suite** shortcut icon ( ) from the Windows or Linux desktop or taskbar.

[Linux users]: If your system has not been configured with a shortcut on the desktop or taskbar, refer to the "Launching the Vivado IDE using a Linux Terminal" section in the *Lab Reference Guide*.

© Copyright 2019 Xilinx

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.

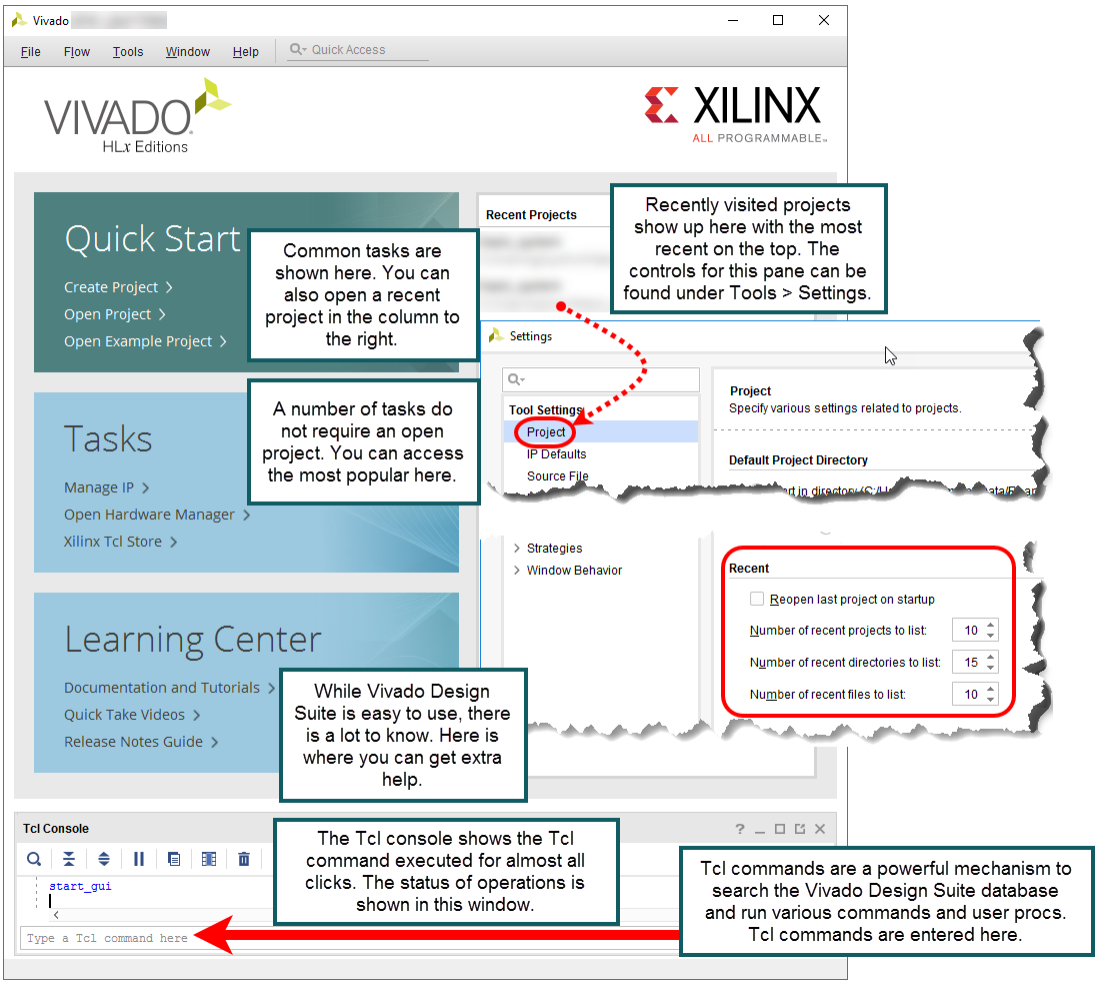


Figure 3-5: Vivado Design Suite Welcome Screen

© Copyright 2019 Xilinx

"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).

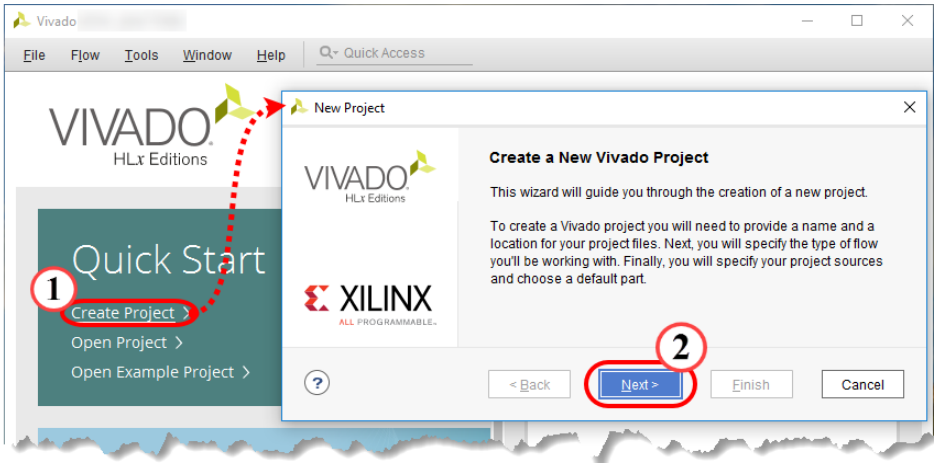


Figure 3-6: Creating a New Vivado Design Suite Project

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).

© Copyright 2019 Xilinx

1-3. You will now encounter a series of dialog boxes asking you to enter different pieces of information describing the project.

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

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

```
$processes/lab/KCU105
```

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 as leaving this checked will create an unnecessary level of hierarchy for this lab (3).

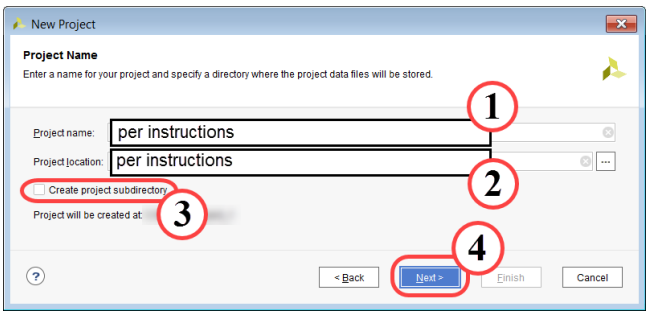


Figure 3-7: Entering the Project Name and Location

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).

1-3-6. Select **Do not specify sources at this time** (2), 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.

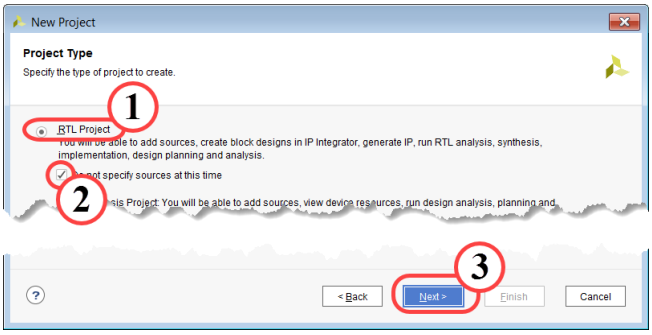


Figure 3-8: Selecting Project Type

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

© Copyright 2019 Xilinx

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. Select **Boards** from the Select 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.

Alternatively you can select the board directly from the list at any time while in this dialog box.

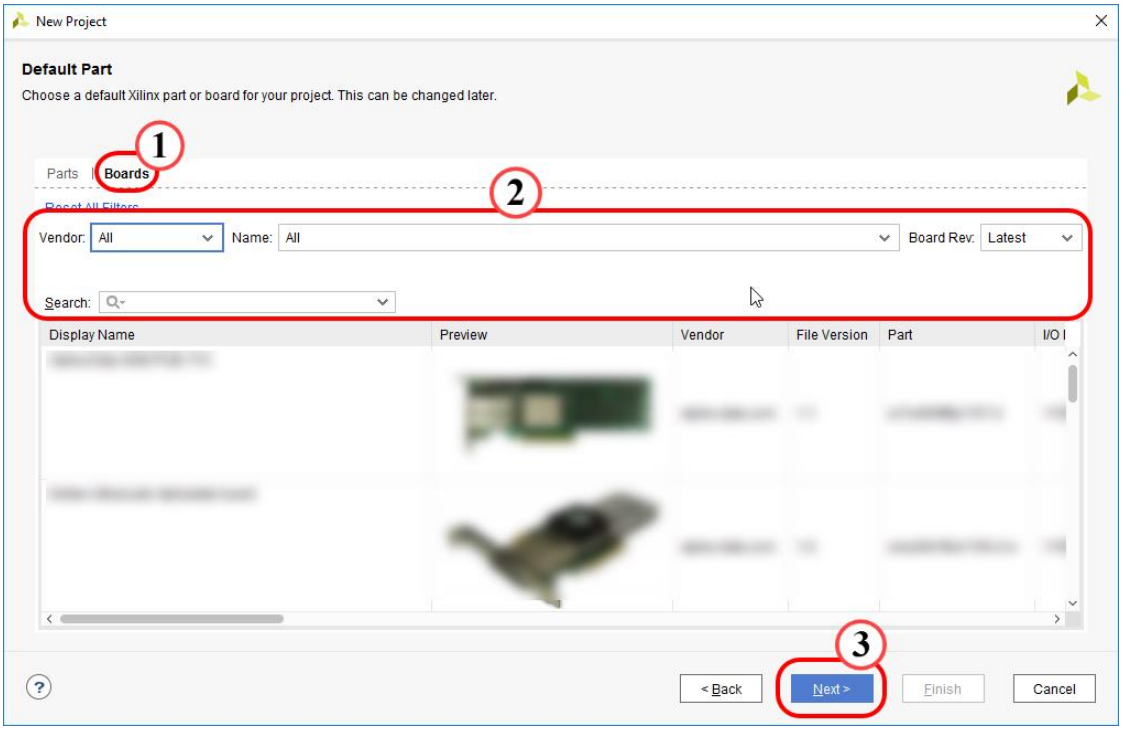


Figure 3-9: Selecting the Board for the Project

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 and making the correction, or you can create the project now and edit the project properties, add or remove files, etc., later.

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 option to **VHDL**.

© Copyright 2019 Xilinx

Creating a New VHDL Source Step 2

Refer to the "Circuit for This Lab" figure (in the Introduction section of this lab) for the coding portion that follows.

You will begin by creating a VHDL module template, then filling the template with a snippet from the Language Templates and filling in the body of the template.

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

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

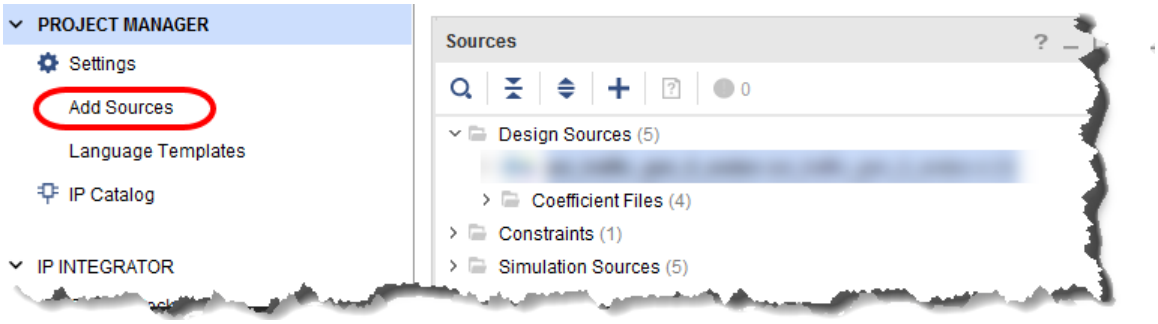


Figure 3-10: Selecting Add Sources

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

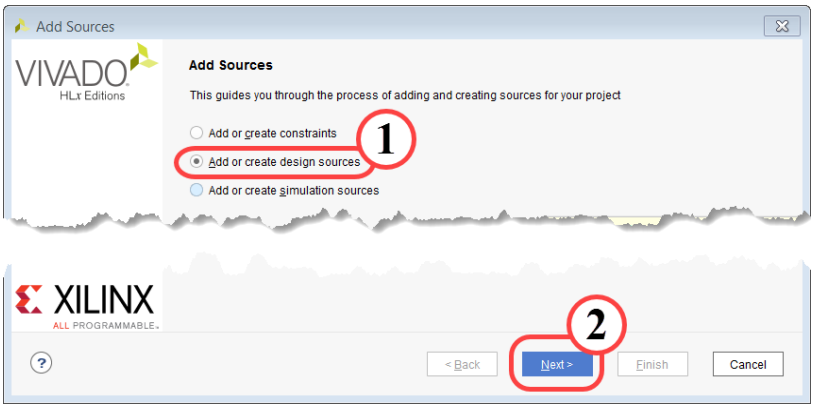


Figure 3-11: Selecting Add or Create Design Sources

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

The Add or Create Design Sources dialog box opens.

© Copyright 2019 Xilinx

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

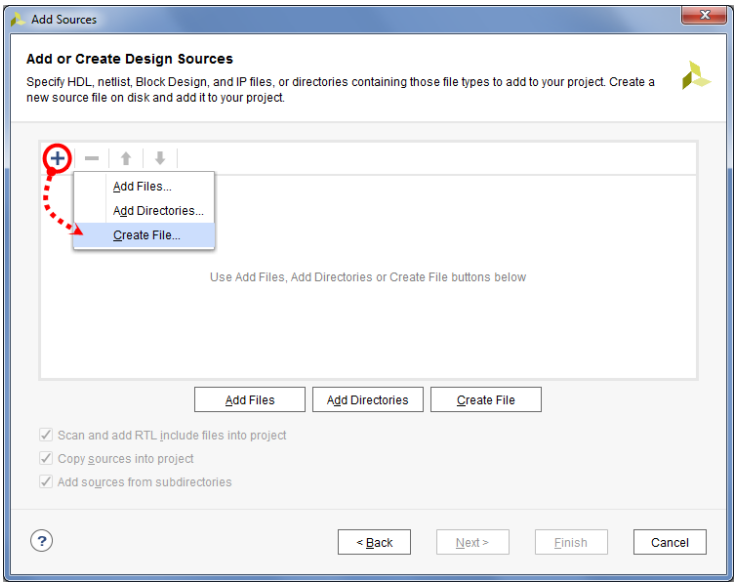


Figure 3-12: Selecting Create File

The Create Source File dialog box opens.

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

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

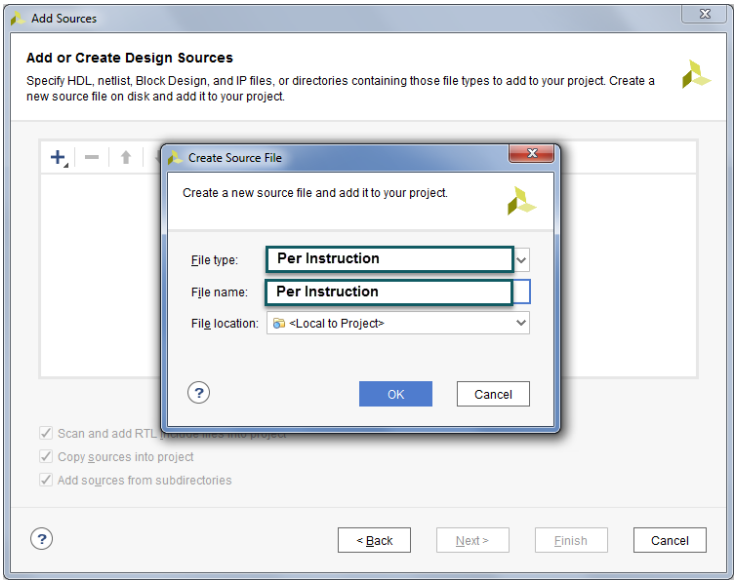


Figure 3-13: Entering File Name and Type

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.

© Copyright 2019 Xilinx

2-2. Specify the following inputs and outputs for the reset_bridge module in the Define Module dialog box.

- **clk_dst – in – std_logic**
- **rst_in – in – std_logic**
- **rst_out – out – std_logic**

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

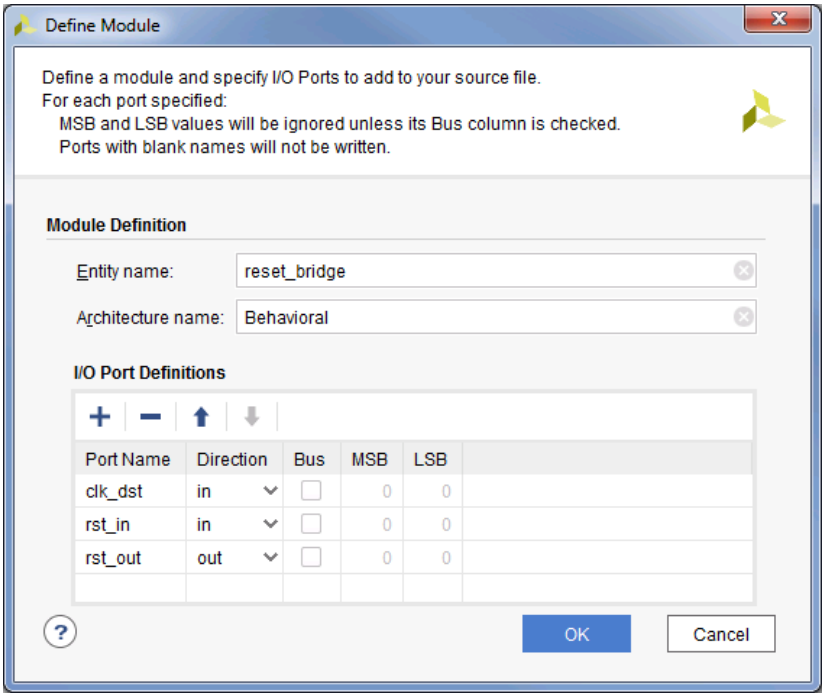


Figure 3-14: Defining Ports for reset_bridge

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

The newly created source file will appear in the Sources view.

Double-click **reset_bridge.vhd**.

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

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

© Copyright 2019 Xilinx

2-4. Describe the behavior for reset_bridge module using Language Templates in the Vivado® IDE.

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

Using the Language Templates, select the snippet for a process clocked on a Posedge and with an asynchronous reset.

2-4-1. Open the Language Templates ( ) under the **Tools** tab.

The Language Templates contain a large number of code snippets that demonstrate how various constructs can be built.

2-4-2. Expand **VHDL** > **Synthesis Constructs** > **Process** > **Posedge Clocked**.

2-4-3. Select **w/Sync Low Reset**.

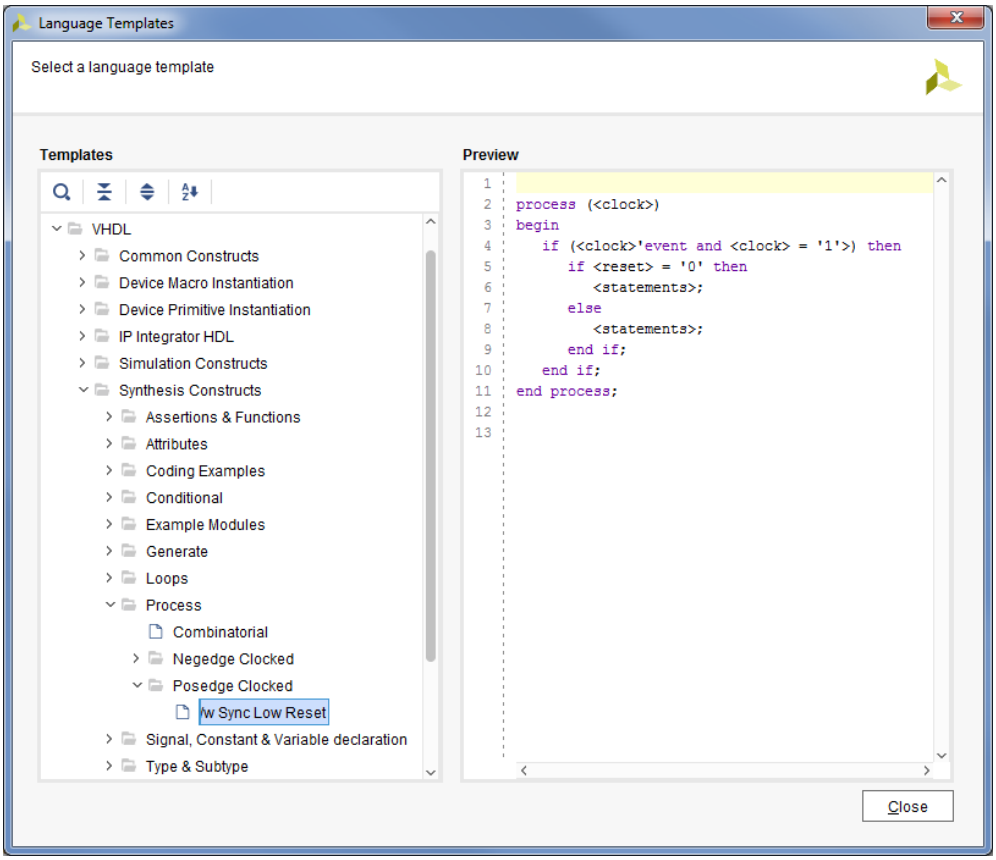


Figure 3-15: Partially Expanded Language Template List

Notice how the code appears in the below window. This code represents a template for creating the object selected from the tree.

© Copyright 2019 Xilinx

2-5. Copy and paste the entire snippet into the source code between the "begin" and "end Behavioral" statements. Indent the snippet.

2-5-1. Select the text in the Preview section of the Language Templates window.

2-5-2. Copy the text by right-clicking the highlighted area and selecting **Copy**.

You can also use the keyboard shortcut **<Ctrl+C>**.

2-5-3. Click the **reset_bridge.vhd** tab to switch back to the source code.

The asterisk means that this file is unsaved.

2-5-4. Click anywhere between the "begin" and "end Behavioral" to position the cursor.

2-5-5. Right-click and select **Paste**.

You can also use the keyboard shortcut **<Ctrl+V>**.

Note the warning in the comments for this snippet. Good design practice calls for intelligent use of white space.

The template is for **Synchronous Low Reset**. Change the code to make **Asynchronous High Reset**.

Your code should look like the figure shown below.

```
40 architecture Behavioral of reset_bridge is
41
42 begin
43
44    process (<clock>,<reset>)
45    begin
46       if <reset> = '1' then
47          <statements>;
48       elsif (<clock>'event and <clock> = '1') then
49          <statements>;
50       end if;
51    end process;
52
53 end Behavioral;
54
```

Figure 3-16: Code Changed to Asynchronous High Reset

© Copyright 2019 Xilinx

2-6. The snippet represents generic code.

Change the items enclosed in the angle brackets with signals that were defined in the entity.

Take your time and examine the code carefully.

Although a suggested code solution can be found in the Answers section, you should make every effort to code this yourself.

The two flip-flops shown in the "Circuit for This Lab" figure in the Introduction of this lab are implied by two signals: the output of the module (*rst_out*) represents the second flip-flop and a variable that becomes the first flip-flop. You will create this variable (named rst_meta).

2-6-1. Make the following substitutions:

- <clock> becomes clk_dst (representing the clock from the destination clock domain)
- <reset> becomes rst_in (representing the raw, uncontrolled active high reset)

Observe line 48 in the above figure. This complex looking line of code means "when a change is detected on the clk_dst signal and that signal is now high…". This translates to "when the rising edge occurs…". To this end, the IEEE has produced several shorthand notations to make this clearer and less error prone.

2-6-2. Replace the conditional portion of the elsif with "rising_edge(clk_dst)".

2-6-3. Declare a variable to represent the wire between two flip-flops.

Because of the way you will use this variable, a flip-flop will be inferred.

Name it "rst_meta" and identify it as a std_logic type.

Remember that variables are declared in the process declaration area, not the architecture declaration area.

2-6-4. Refer back to the "Circuit for this Lab" figure. Notice the behavior of the flip-flops upon application of an asynchronous reset. The flip-flops are driven to logic high. Add this behavior to the <statements> area following the test for reset high.

2-6-5. Refer back to the "Circuit for this Lab" figure and feed the output of the first flip-flop to the input of the second flip-flop after the test for the rising edge of the clock (that is, the synchronous events).

The output of the second flip-flop should provide the output rst_out.

Hint: Recall that variables like rst_meta take on their value immediately, whereas signals take on their assignment when the process suspends in this case, at the process sensitivity list. This implies that the ordering of the signal assignments will change the behavior of the code.

© Copyright 2019 Xilinx

Question 1

What would be wrong if the code were written with the variable assignment preceding the signal assignment?

_____

_____

_____

2-6-6. Correct the code to place the assignment to rst_meta AFTER that of rst_out.

2-6-7. Select **File** > **Text Editor** > **Save File**.

2-6-8. Close the Language template view by clicking '**X**' in the top-right of the Language Templates view.

Verifying with the Viewer and Simulating — Step 3

Now that the code has been entered, you must verify that it is what you intended.

3-1. Run synthesis.

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

3-1-2. Click **OK** to launch the runs.

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

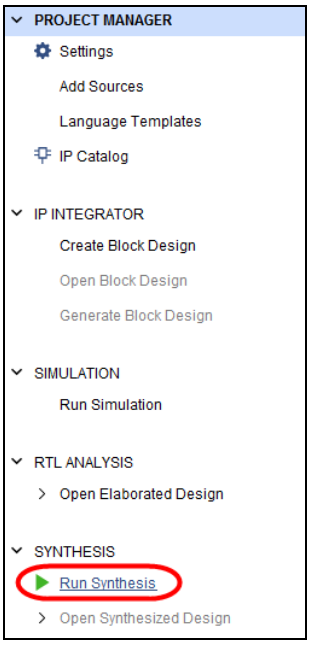


Figure 3-17: Selecting Run Synthesis

© Copyright 2019 Xilinx

3-1-3. Click **Save** if you are asked to save your files.

After the synthesis process completes, the Synthesis Completed dialog box opens. The dialog box prompts you to run implementation, open the synthesized design, or view reports.

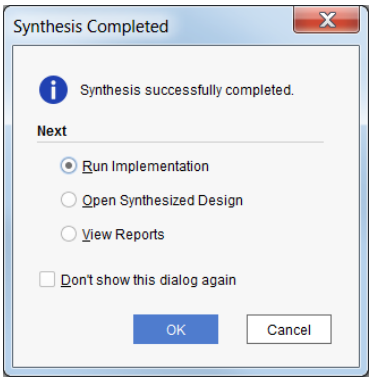


Figure 3-18: Synthesis Completed Dialog Box

3-1-4. Click **OK** to continue with your preferred choice or **Cancel** to simply close the dialog box and return to the normal view of the Vivado Design Suite.

3-2. Use the Schematic viewer to explore the design.

The Schematic viewer shows the logic and register relationships and how it is implemented in the fabric.

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

3-2-1. Open the synthesized design and right-click **reset_bridge** in the Netlist view and select **Schematic**.

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

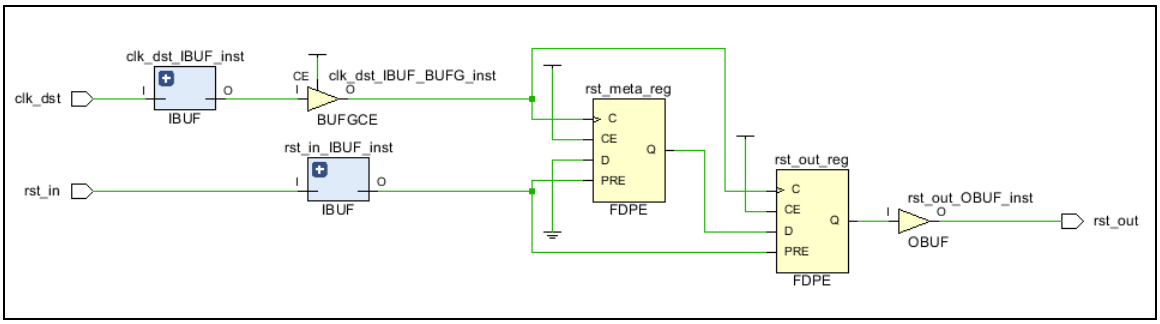


Figure 3-19: Final Design (KCU105)

© Copyright 2019 Xilinx

Question 2

Does the final design in the Schematic view appear as you would expect? Why or why not?

_____

_____

_____

3-2-2. Close the synthesized design.

3-3. Because the Schematic viewer suggests that what was built is correct, it is now time to perform behavioral verification with simulation.

Add the provided *reset_bridge_tb.vhd* testbench to the project. Examine the testbench and note how the various signal stimuli are generated.

3-3-1. Click **Add Sources** in the Flow Navigator under Project Manager.

3-3-2. Select **Add or Create Simulation Sources** and click **Next**.

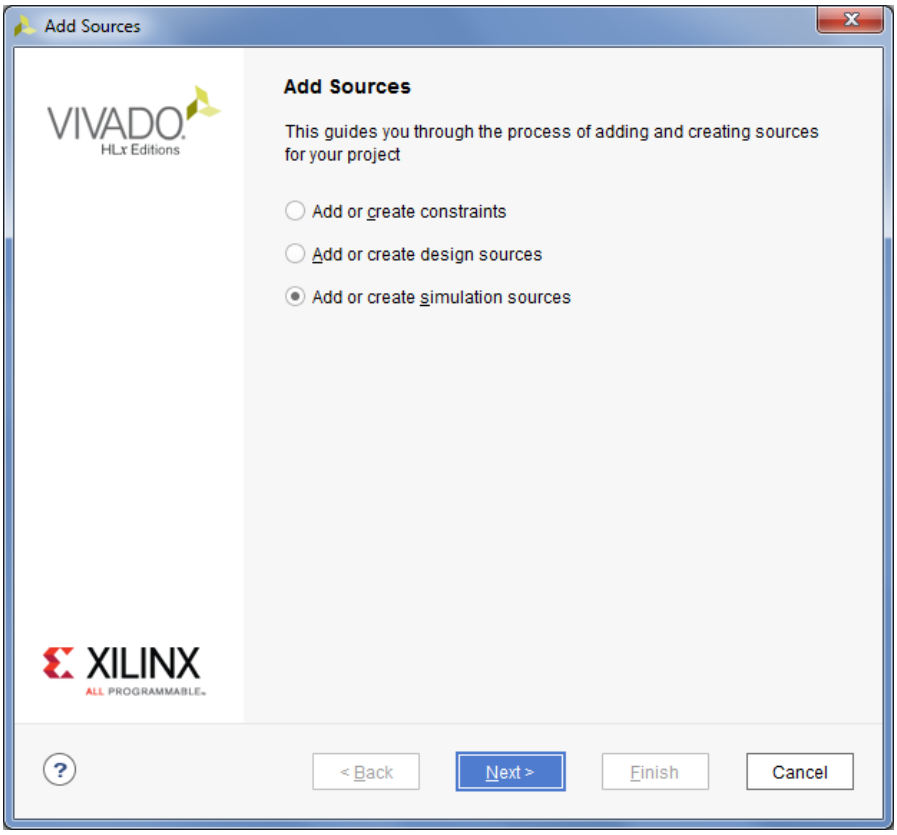


Figure 3-20: Add Sources for Simulation

The Add or Create Simulation Sources dialog box opens.

© Copyright 2019 Xilinx

3-3-3. Click **Add Files**.

3-3-4. Select **reset_bridge_tb.vhd** and click **OK**.

3-3-5. Click **Finish** to confirm adding the selected file to the project.

3-4. Explore the testbench to see how the various signal stimuli are generated.

3-4-1. Expand **Simulation Sources** > **sim_1** in the Sources view.

The simulation sources files are listed with the design files present in the hierarchy.

3-4-2. Double-click **reset_bridge_tb.vhd** in the Sources view under Simulation Sources to open the file in the text editor.

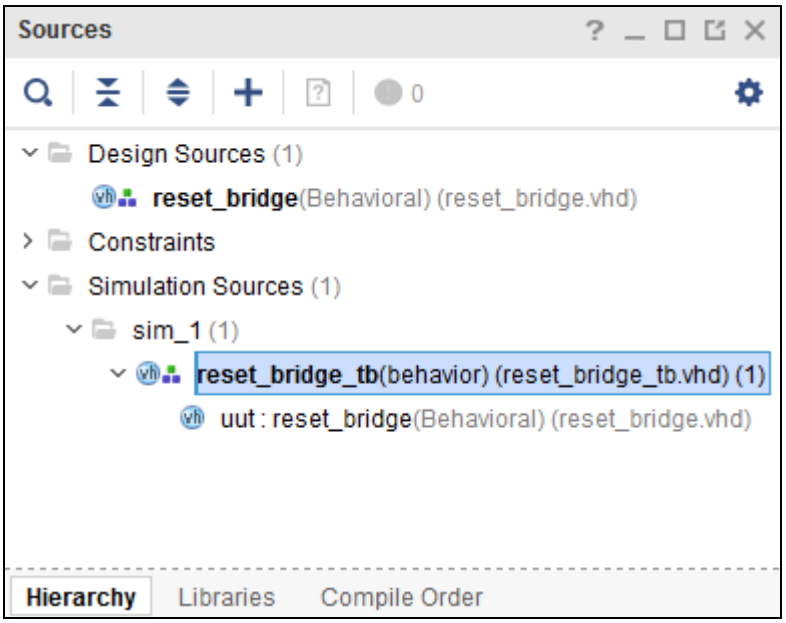


Figure 3-21: Opening the Testbench

Note how the various signal stimuli are generated.

© Copyright 2019 Xilinx

3-5. Launch the simulation for reset_bridge_tb.

3-5-1. Select **Simulation** > **Run Simulation** in the Flow Navigator.

3-5-2. Select **Run Behavioral Simulation**.

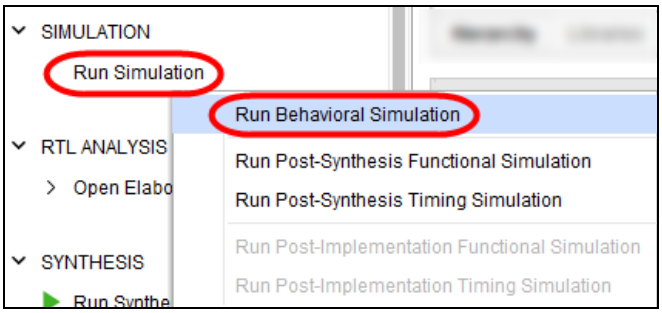


Figure 3-22: Launching the Vivado Simulator Tool

This will launch the Vivado Simulator.

3-6. Run the simulation for 1 µs and view the full waveform.

3-6-1. Select **Run** > **Restart** (or click the icon) to reset the simulation.

3-6-2. Enter **1 us** in the duration field.

Figure 3-23: Vivado Simulator Control Toolbar

3-6-3. Click the icon to run the simulation for 1 us.

3-6-4. Click the **Zoom Fit** ( ) icon in the horizontal toolbar of the waveform viewer to zoom to full screen.

Question 3

Is the design working?

_____

_____

_____

Question 4

What happens when the reset and clock are asserted simultaneously?

_____

_____

© Copyright 2019 Xilinx

3-7. Close the simulation.

3-8. Close the Vivado Design Suite.

3-8-1. Select **File** > **Exit**.

The Exit Vivado dialog box opens.

Figure 3-24: Exit Vivado Dialog Box

3-8-2. Click **OK**.

[Optional]: Clean up the file system.

Some systems 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. Both processes will recursively delete all of the files in the `$processes` directory.

3-9. Clean up the VirtualBox file system via the GUI.

3-9-1. Using the graphical browser (Windows: press the <**Windows**> key + <**E**>; Linux: press <**Ctrl** + **N**>), navigate to `$processes`.

3-9-2. Select **`processes`**.

3-9-3. Press <**Delete**>.

3-10. Clean up the VirtualBox file system via the command line.

3-10-1. Open a terminal window (Windows: press the <**Windows**> key + <**R**>, then enter **cmd**; Linux: press <**Ctrl** + **Alt** + **T**>)

3-10-2. Enter the following command to delete the contents of the workspace:

[Windows users]: **`rd /s /q $processes`**

[Linux users]: **`rm -rf $processes`**

© Copyright 2019 Xilinx

Summary

This lab introduced a commonly used design for asserting resets asynchronously and deasserting them synchronously while guaranteeing that the reset pulse lasts at least one clock cycle. If you add a shift register or counter, the reset pulse width can be expanded as you see fit.

This lab also showed how a simple process can be constructed to encapsulate sequential statements and illustrates the importance of when signals and variables take on their values.

© Copyright 2019 Xilinx

Answers

1. What would be wrong if the code were written with the variable assignment preceding the signal assignment?

   One register with an enable, clock, data in, data out, and a reset would be built because the rst_meta signal would simply rename a wire leading from ground to the reset output.

   This is not what you want. Placing the rst_meta assignment AFTER the signal assignment will infer a register.

2. Does the final design in the Schematic view appear as you would expect? Why or why not?

   Recall that variables take on their value immediately. So, if the rst_meta variable is assigned BEFORE rst_out gets rst_meta, then meta_rst will take on its value immediately and will be assigned to rst_out—thereby effectively "disappearing."

   Placing the assignment to rst_meta AFTER it is assigned to rst_out creates a causal (time dependent) loop and thereby creates the desired flop.

3. Is the design working?

   For this level of simulation, yes, the logic behaves correctly.

4. What happens when the reset and clock are asserted simultaneously?

   Meta-stability is not modeled for behavioral-level simulation because the physical parameters of the gates are not known. When reset is asserted simultaneously with the clock, then it is considered to rise instantly and has 0 setup and hold time.

© Copyright 2019 Xilinx

Coding the Process

Making the following substitutions:

- <clock> becomes clk_dst (representing the clock from the destination clock domain)
- <reset> becomes rst_in (representing the raw, uncontrolled active high reset)

```
30  architecture Behavioral of reset_bridge is
31
32     begin
33
34        process (clk_dst,rst_in)
35        begin
36           if rst_in = '1' then
37              <statements>;
38           elsif (clk_dst'event and clk_dst = '1') then
39              <statements>;
40           end if;
41        end process;
42
43     end Behavioral;
44
```

Declaring a variable named "rst_meta" to represent the wire between two flip-flops:

```
34        process (clk_dst,rst_in)
35           variable rst_meta : std_logic := 'U';
36        begin
37           if rst_in = '1' then
```

Adding behavior in which the flip-flops are driven to logic high upon application of an asynchronous reset:

```
34        process (clk_dst,rst_in)
35           variable rst_meta : std_logic := 'U';
36        begin
37           if rst_in = '1' then
38              rst_meta := '1';      -- the meta-stable likely flop is driven high internal to the process
39              rst_out  <= '1';      -- the output of the module is also driven high
40           elsif (rising_edge(clk_dst)) then
```

Feeding the output of the first flip-flop to the input of the second flip-flop after the test for the rising edge of the clock:

```
34        process (clk_dst,rst_in)
35           variable rst_meta : std_logic := 'U';
36        begin
37           if rst_in = '1' then
38              rst_meta := '1';      -- the meta-stable likely flop is driven high internal to the process
39              rst_out  <= '1';      -- the output of the module is also driven high
40           elsif (rising_edge(clk_dst)) then
41              rst_meta := '0';      -- clear the meta-stable likely flop
42              rst_out <= rst_meta; -- feed the second flop the output of the first flop
43           end if;
44        end process;
```

Placing the assignment to rst_meta AFTER that of rst_out:

```
34        process (clk_dst,rst_in)
35           variable rst_meta : std_logic := 'U';
36        begin
37           if rst_in = '1' then
38              rst_meta := '1';      -- the meta-stable likely flop is driven high internal to the process
39              rst_out  <= '1';      -- the output of the module is also driven high
40           elsif (rising_edge(clk_dst)) then
41              rst_out <= rst_meta; -- feed the second flop the output of the first flop
42              rst_meta := '0';      -- clear the meta-stable likely flop
43           end if;
44        end process;
```


© Copyright 2019 Xilinx

Lab 4: Building a Dual-Port Memory

2019.1

Abstract

This lab illustrates how to design a dual-port memory in VHDL. It should take approximately 60 minutes.

Objectives

After completing this lab, you will be able to:

- Demonstrate how memory can be inferred
- Guide the inference to specific types of implementations

Introduction

The wave_gen design, used in this lab, uses a dual-port block RAM memory to store incoming patterns from one side and play these patterns back out on the other side. The wave_gen design requires that the "A" side of the dual-port memory have read and write capabilities, while the "B" side only needs to read from the device.

The wave_gen design contains a dual-port memory that was created using the Block Memory Generator IP. While it is a capable tool, using the output of this tool locks a designer into a single family and the core must be regenerated in order to port the design to other devices. Inferring dual-port memory removes this hindrance.

Conveniently, both distributed and block RAM implementations of dual-port memory inference support a simple dual-port configuration (read and write on one side, read only on the other).

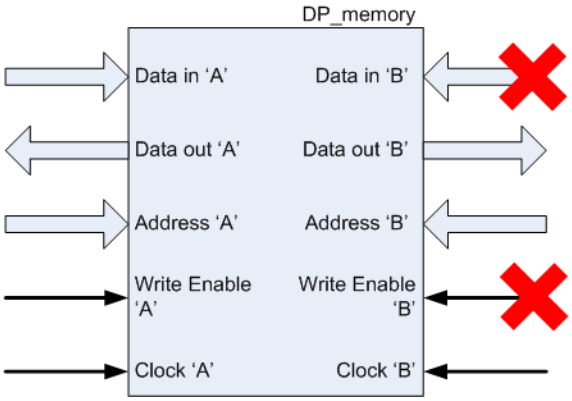


Figure 4-1: Block Memory Block Diagram

The source code that you will write will support the full reading and writing of the DP_memory and will be tested using a pre-written testbench.

© Copyright 2019 Xilinx

You will then comment out the items marked with the 'X' in the figure above and synthesize, using block RAM resources.

Understanding the Lab Environment

The labs and demos provided in this course are designed to run under the provided Linux platform. The provided Linux platform has a customized lab environment. These customizations have been made to simplify and enhance your learning experience. Many labs and demos can be successfully executed in the Windows environment as well. Topics including PetaLinux, Yocto, and QEMU are generally not supported under Windows.

The instructions, for the sake of brevity, are expressed using the Linux notation. This includes the forward slash ('/') as the hierarchy separator instead of the Windows backslash ('\'). For students who want to run under Windows, it is their responsibility to use the correct hierarchy separator.

Windows users can run Linux using the VirtualBox tool from Oracle. Details on how to acquire and install this tool can be found in the lab setup guide. The Xilinx Customer Education group offers a preconfigured virtual machine image and includes all the tools necessary to run the labs. Some tools and IP do require licenses, which are the responsibility of the student.

Several environment variables, described in detail in the lab setup guide, which also discusses how you can customize the environment for your situation, are provided for ease of use. Most notably, many environment variables have been defined to reduce the amount of typing you need to do when entering paths.

These environment variables are:

- XILINX_PATH - points to the installation directory for the Xilinx tools. Tools such as the Vivado Design Suite and SDK can be found here.
- PETALINUX_PATH - points to the installation directory for the Xilinx PetaLinux tools. Often this is within XILINX_PATH, but some choose to install it elsewhere.
- TRAINING_PATH - points to the space allocated for students to work through their labs. This directory includes prebuilt images and starting points for the labs and demos.
- buildingMemory - points to the specific name of the topic within the TRAINING_PATH directory.

  This lab uses $buildingMemory to indicate the directory for this lab, which is equivalent to $TRAINING_PATH/buildingMemory.

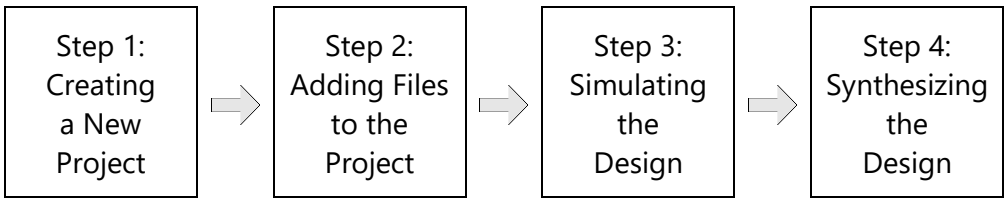
These environment variables work well in the Linux terminal and the Vivado Design Suite and scripts, but not in Eclipse-based tools such as the SDK tool and the SDx™ environment. When using the Vivado tools, you can use $buildingMemory and the tools will automatically expand the environment variable for you.

However, when you see the notation $buildingMemory/workspace in SDK or the SDx environment, you should interpret it in its expanded form. The expanded form can be found by opening a terminal window and enter "echo $buildingMemory".

© Copyright 2019 Xilinx

This lab assumes that you are starting with a Linux environment. If you are working on a Windows system, refer to the lab setup guide to see how to use the Oracle VirtualBox application to run the Xilinx Customer Education pre-configured Ubuntu Linux virtual machine.

General Flow



Creating a New Project Step 1

1-1. Launch the Vivado Design Suite.

If you do not recall how to perform this task, refer to the "Launching the Vivado Design Suite" section under Vivado Design Suite Operations in the *Lab Reference Guide*.

1-2. Create a new Vivado Design Suite project named *buildingMemory* and locate it in the following directory.

Browse to the `$buildingMemory/lab/KCU105` directory.

Target the project for the Kintex UltraScale KCU105 Evaluation Board evaluation board.

If you do not recall how to perform this task, refer to the "Creating a Blank Vivado Design Suite Project" section in the *Lab Reference Guide*.

From the settings in the Flow Navigator, change the target language to VHDL.

© Copyright 2019 Xilinx

Adding Files to the Project Step 2

You will be coding the bulk of this design based on the I/O specifications. The only file that needs to be imported is the testbench and Vivado® simulator wave configuration file.

2-1. Load the DPmemory_tb.vhd testbench file into the project. It has been preloaded into the working directory.

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

The Add Sources Wizard opens.

2-1-2. Select **Add or create simulation sources**.

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

2-1-4. Click the **Plus** (+) icon and select **Add Files**.

2-1-5. Select **DPmemory_tb.vhd** and click **OK**.

2-1-6. Click **Finish** to confirm adding the selected file to the project.

2-2. Create a new HDL source file called *DP_memory*.

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

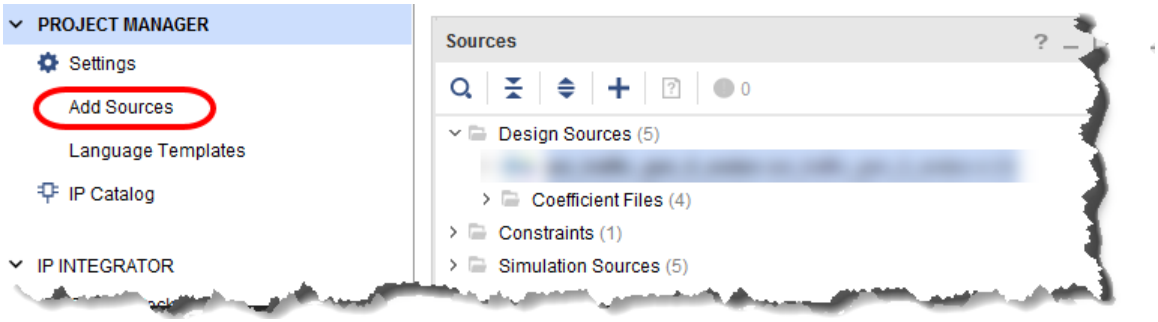


Figure 4-2: Selecting Add Sources

© Copyright 2019 Xilinx

2-2-2. Select **Add or create design sources**.

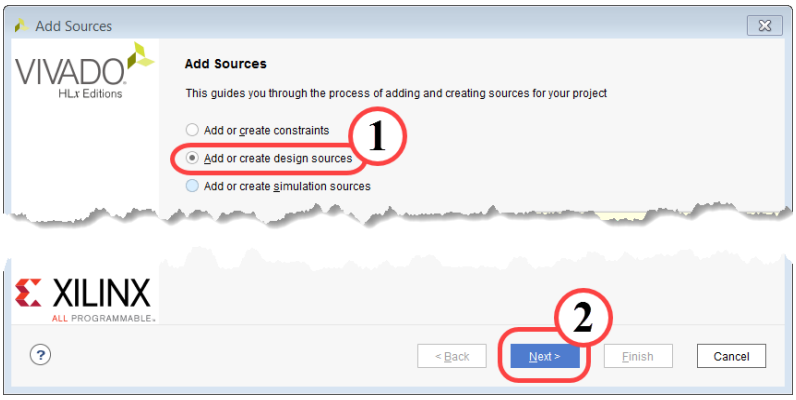


Figure 4-3: Selecting Add or Create Design Sources

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

The Add or Create Design Sources dialog box opens.

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

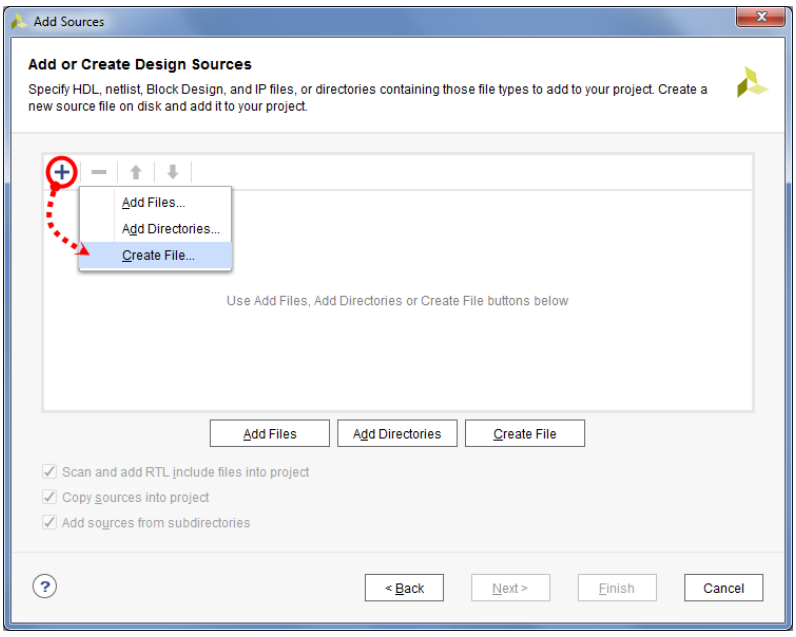


Figure 4-4: Selecting Create File

The Create Source File dialog box opens.

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

© Copyright 2019 Xilinx

2-2-6. Enter **DP_memory** as the file name.

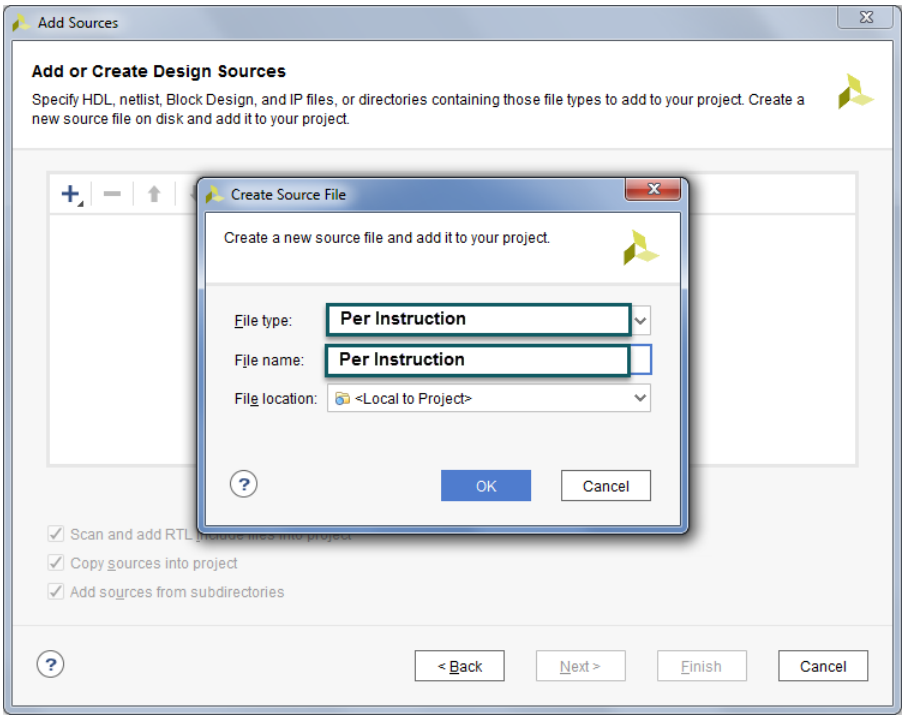


Figure 4-5: Entering File Name and Type

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

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

The Define Module dialog box opens.

2-3. Specify the following input and output ports for the DP_Memory module in the Define Module dialog box.

Signal Name	Direction	Type
PortA_clk	In	std_logic
PortA_addr	In	std_logic_vector(9 downto 0)
PortA_dataIn	In	std_logic_vector(7 downto 0)
PortA_writeEnable	In	std_logic
PortA_dataOut	Out	std_logic_vector(7 downto 0)
PortB_clk	In	std_logic
PortB_addr	In	std_logic_vector(9 downto 0)

© Copyright 2019 Xilinx

Signal Name	Direction	Type
PortB_dataIn	In	std_logic_vector(7 downto 0)
PortB_writeEnable	In	std_logic
PortB_dataOut	Out	std_logic_vector(7 downto 0)

DP_memory Inputs and Outputs

2-3-1. Add the ports according to the DP_memory Inputs and Outputs table.

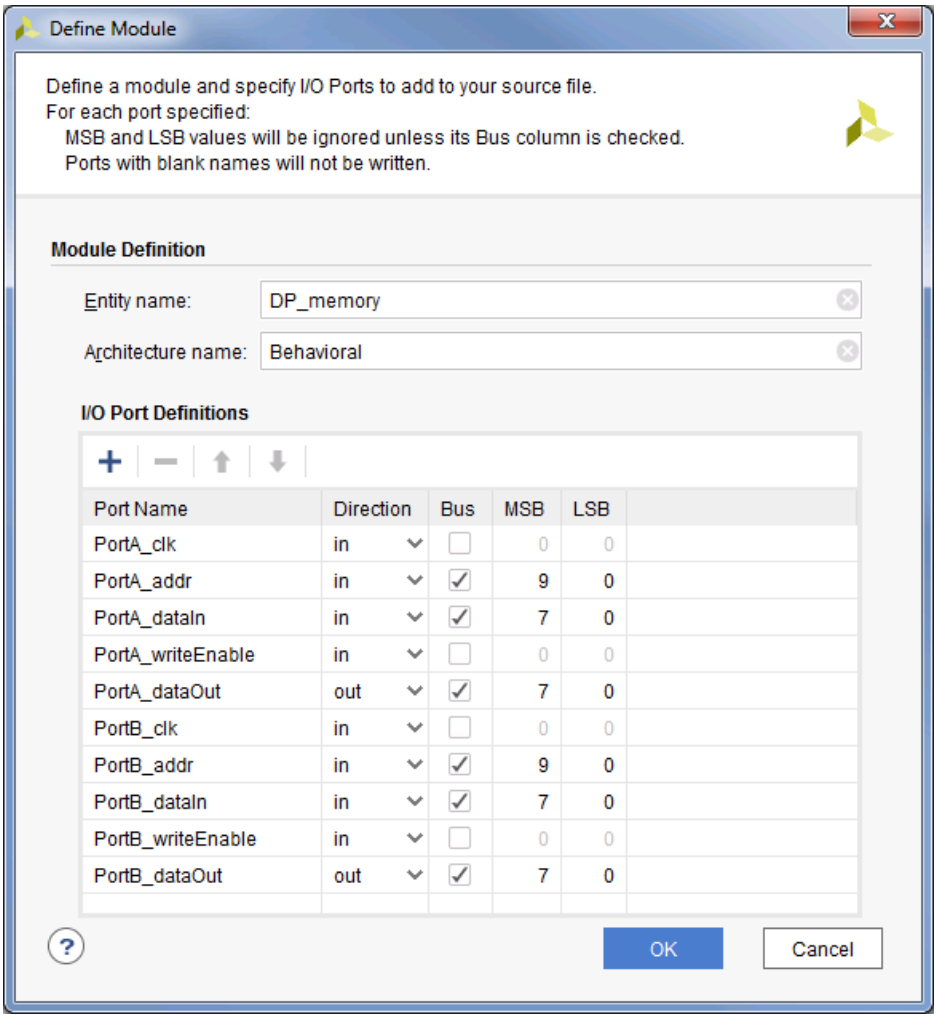


Figure 4-6: Adding Ports to the New Module

2-3-2. Click **OK**.

© Copyright 2019 Xilinx

2-4. Uncomment the IEEE.NUMERIC_STD package in the code.

2-4-1. Double-click the **DP_memory.vhd** file to make the source code visible in the main workspace window.

2-4-2. Remove the **- -** comment symbols before the **use IEEE.NUMERIC_STD.ALL;** on line 27 to include the NUMERIC_STD package.

2-5. Create a memory structure that is 1024 x 8 bits in size and an appropriate shared variable to represent the memory itself.

There are a number of significant side effects that can be caused by the misuse of shared variables; however, as in this case, it is the only way to properly model a dual-port memory. Suffice it to say that shared variables behave exactly like regular variables, but their scope is limited to the module that they reside in rather than just the process.

Although a suggested code solution can be found in the Answers section, you should make every effort to code this yourself.

2-5-1. Add the code (type statement) for a memory structure that is 1024 x 8 bits in size prior to the architecture's "begin" statement and name it memoryByteArray.

2-5-2. Add an appropriate shared variable named memoryBlock for the memory itself immediately after the type statement.

Shared variables are declared with the key words "shared variable" instead of variable or signal and are used to share data between processes. There are a number of considerations to be made when using shared variables.

2-6. Create two processes – one for managing the 'A' side of the memory, the other for the 'B' side of memory. The processes will be identical except for the signals that they will be working on.

The process should be sensitive to the rising edge of the appropriate clock. Assign data_in to the memory structure when the write enable is asserted, and always present the data referenced by the address to the data output.

Although suggested code solutions can be found in the Answers section, you should make every effort to code these yourself.

2-6-1. Construct a process that is sensitive to the rising edge of the PortA_clk immediately following the "begin" statement of the architecture.

Because VHDL uses integers as the index into arrays, the address must be converted from std_logic_vector to an integer.

2-6-2. Create a temporary variable named "address" as an integer of the proper size prior to the process's "begin" statement to act as an index in the 1024x8 memory.

© Copyright 2019 Xilinx

2-6-3. Perform the conversion on the std_logic_vector version of the address into the variable address within the clocked process.

2-6-4. Add the appropriate code within the synchronous portion of the process so that the memory must always present the contents pointed to by the address on the data_out port.

The last step is to provide the write capability to the process.

2-6-5. Add code that tests the enable to determine if the write should occur, and if it should, perform the write operation.

2-6-6. Repeat this block of code, but for the 'B' side of the memory in another process.

2-6-7. Select **File** > **Text Editor** > **Save File**.

The final code should appear as in the Answers section.

2-7. Synthesize the design. Correct any errors.

2-7-1. Click **Run Synthesis** in the Flow Navigator and click **OK**.

2-7-2. Correct any errors that have been found and resynthesize.

2-7-3. Click **Cancel** when the Synthesis Completed dialog box appears.

Simulating the Design Step 3

This portion of this lab requires you to load a predefined configuration for the simulation environment, run the simulation, and view the waveform.

3-1. Launch the provided testbench and note the order and appearance of the signals.

3-1-1. Select **Simulation** > **Run Simulation** in the Flow Navigator.

3-1-2. Select **Run Behavioral Simulation**.

This will launch the Vivado simulator and run for the default 1 µs. Note the order and radices of the signals.

3-2. Load the provided configuration file DPmemory.wcfg in the Vivado simulator.

3-2-1. Select **File** > **Simulation Waveform** > **Open Configuration**.

3-2-2. Select **DPmemory.wcfg** in the `$buildingMemory/lab/KCU105` directory and click **OK**.

© Copyright 2019 Xilinx

Question 1

What changed when the configuration file was loaded?

_____

_____

_____

3-3. Examine the waveform, review the testbench, and answer the questions below.

Question 2

What occurs during the first 50 ns of the simulation?

_____

_____

_____

Question 3

What occurs at approximately 555 ns and why is this done?

_____

_____

_____

Question 4

What occurs after 570 ns and what does this show?

_____

_____

_____

3-4. Exit the Vivado simulator.

3-4-1. Select **File** > **Close Simulation**.

3-4-2. Click **OK** to close the simulation.

© Copyright 2019 Xilinx

Synthesizing the Design Step 4

Now that you know that the code is operating as expected, you will remove the write capability from the 'B' side so that the functionality for the wave_gen design is met.

4-1. Return to the source file and edit the DP_memory.vhd file so that all references to writing to the Port B side of block RAM is removed. This will implement as a simple dual-port memory rather than a true dual-port memory.

Use comments rather than deleting the lines.

4-1-1. Comment out PortB_writeEnable and PortB_dataIn from the port declaration in the entity statement.

```
36  entity DP_memory is
37      Port ( PortA_clk : in STD_LOGIC;
38             PortA_addr : in STD_LOGIC_VECTOR (9 downto 0);
39             PortA_dataIn : in STD_LOGIC_VECTOR (7 downto 0);
40             PortA_writeEnable : in STD_LOGIC;
41             PortA_dataOut : out STD_LOGIC_VECTOR (7 downto 0);
42             PortB_clk : in STD_LOGIC;
43             PortB_addr : in STD_LOGIC_VECTOR (9 downto 0);
44  --           PortB_dataIn : in STD_LOGIC_VECTOR (7 downto 0);
45  --           PortB_writeEnable : in STD_LOGIC;
46             PortB_dataOut : out STD_LOGIC_VECTOR (7 downto 0));
47  end DP_memory;
```

Figure 4-7: Port B Write Capability Commented Out of the Entity

4-1-2. Remove the references for PortB_writeEnable and PortB_dataIn from the PortB process.

To save time, highlight the block of code and use the block comment capability by clicking the icon.

```
66
67              PortB: process(portB_clk)
68             variable address: integer range 0 to 1023:=0;
69           begin
70
71              if rising_edge(PortB_clk) then
72                  address:= to_integer(unsigned(portB_addr));
73  --                if(portB_writeEnable='1') then
74  --                 memoryBlock(address):=PortB_dataIn;
75  --                end if;
76                  portB_dataOut<= memoryBlock(address);
77              end if;
78        end process PortB;
```

Figure 4-8: Port B Write Capability Commented Out of the Process

4-1-3. Select **File** > **Text Editor** > **Save File**.

© Copyright 2019 Xilinx

4-2. Run synthesis.

4-2-1. Click **Run Synthesis** in the Flow Navigator under Synthesis.

4-2-2. Click **OK** to launch the runs.

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

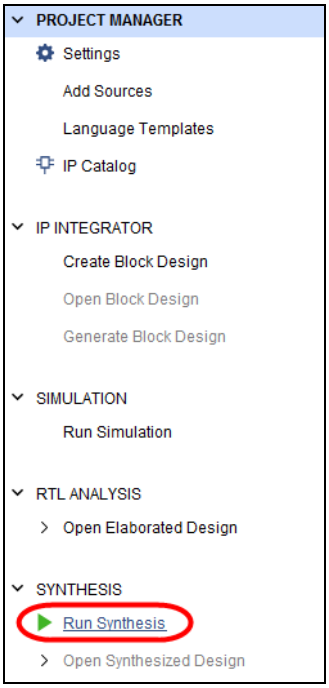


Figure 4-9: Selecting Run Synthesis

4-2-3. Click **Save** if you are asked to save your files.

After the synthesis process completes, the Synthesis Completed dialog box opens. The dialog box prompts you to run implementation, open the synthesized design, or view reports.

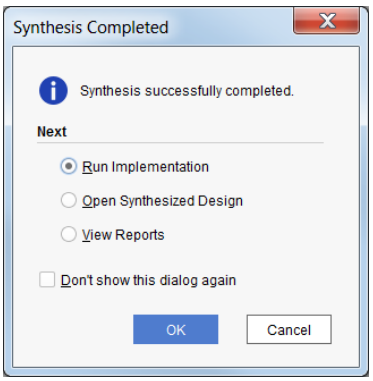


Figure 4-10: Synthesis Completed Dialog Box

4-2-4. Click **OK** to continue with your preferred choice or **Cancel** to simply close the dialog box and return to the normal view of the Vivado Design Suite.

© Copyright 2019 Xilinx

4-3. Generate a Utilization report.

4-3-1. Click **Report Utilization** under Synthesized Design in the Flow Navigator.

Alternatively, you can select **Reports** > **Report Utilization**.

4-3-2. Click **OK** to accept the default name for the utilization report.

4-3-3. View the Utilization Report that appears at the bottom of the Vivado IDE and complete the Utilization Count column in the Resources table below.

Question 5

Complete the table below.

Resource	Utilization Count
Number of Slice Registers	
Number of Slice LUTs	
Number of Block RAMs	
Number of Bonded IOBs	

Resources Table

4-4. Close the project.

4-5. Close the Vivado Design Suite.

[Optional]: Clean up the file system.

Some systems 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. Both processes will recursively delete all of the files in the `$buildingMemory` directory.

4-6. Clean up the VirtualBox file system via the GUI.

4-6-1. Using the graphical browser (Windows: press the <**Windows**> key + <**E**>; Linux: press <**Ctrl** + **N**>), navigate to `$buildingMemory`.

4-6-2. Select **`buildingMemory`**.

4-6-3. Press <**Delete**>.

© Copyright 2019 Xilinx

4-7. Clean up the VirtualBox file system via the command line.

4-7-1. Open a terminal window (Windows: press the <**Windows**> key + <**R**>, then enter **cmd**; Linux: press <**Ctrl** + **Alt** + **T**>)

4-7-2. Enter the following command to delete the contents of the workspace:

[Windows users]: `rd /s /q $buildingMemory`

[Linux users]: `rm -rf $buildingMemory`

Summary

This lab showed you how to create and simulate a dual-port RAM. After the design was "proven," you removed the B-side write access for two reasons: the design did not require it, and a full dual-port memory can only be generated with a block RAM.

© Copyright 2019 Xilinx

Answers

Answers listed represent sample solutions only. Your results may differ depending on the version of the software, service pack, or operating system that you are using.

1. What changed when the configuration file was loaded?

   The order of the signals was re-arranged so that all the Port A signals are collected together and all the Port B signals are collected together. Additionally, the radix changed on all of the std_logic_vector signals to hexadecimal for ease of reading and two dividers appeared to help delineate which signals belong to Port A and which to Port B.

2. What occurs during the first 50 ns of the simulation?

   Port A is being exercised by writing 0 to address 0, 11 hex to address 1, 22 hex to address 2, and so on, up to and including address 4.

   Simultaneously, a similar process is occurring on Port B. 44 is being written to address 4 and so on, up to address 8.

3. What occurs at approximately 555 ns and why is this done?

   Via Port B, the value of EE hex over-writes address 4. This is done to see if this value appears when the Port A side is read, thus proving that the memory is properly shared.

4. What occurs after 570 ns and what does this show?

   After 570 ns, both Ports A and B begin cycling through the lowest addresses to see if the values are properly stored and re-read.

5. Complete the table below.

Resource	Utilization Count
Number of Slice Registers	0
Number of Slice LUTs	0
Number of Block RAMs	1
Number of Bonded IOBs	47

Resources Table

Note: To get the block RAM utilization count, click the Block RAM Tile under BLOCKRAM in the left pane of utilization report and choose DP_memory. The count will be shown in the Netlist Properties window.

© Copyright 2019 Xilinx

Coding the Memory Structure

Memory structure that is 1024 x 8 bits in size and an appropriate shared variable for the memory itself:

```
type memoryByteArray is array(0 to 1023) of std_logic_vector(7 downto 0);

shared variable memoryBlock : memoryByteArray := (others=>(others=>'0'));
```

Process that is sensitive to the rising edge of the PortA_clk:

```
PortA: process (PortA_clk)
    begin
        if rising_edge(PortA_clk) then
        end if;
    end process PortA;
```

Variable named "address" as an integer of the proper size to act as an index into the 1024x8 memory:

```
variable address : integer range 0 to 1023 := 0;
```

Performing the conversion on the address:

```
address := to_integer(unsigned(PortA_addr));
```

Presenting the contents pointed to by the address on the data_out port:

```
PortA_dataOut <= memoryBlock(address);
```


© Copyright 2019 Xilinx

Testing the enable to determine if the write should occur, and if it should, performing the write operation:

```
if (PortA_writeEnable = '1') then
   memoryBlock(address) := PortA_dataIn;
end if;
```

```
69
70  architecture Behavioral of DP_memory is
71      type memoryByteArray is array(0 to 1023) of std_logic_vector(7 downto 0);
72      shared variable memoryBlock : memoryByteArray:=(others =>(others=> '0'));
73  begin
74
75     PortA: process(portA_clk)
76            variable address : integer range 0 to 1023 := 0;
77          begin
78
79          if rising_edge(PortA_clk) then
80              address:= to_integer(unsigned(portA_addr));
81              if(portA_writeEnable='1') then
82                memoryBlock(address):=PortA_dataIn;
83              end if;
84              portA_dataOut<= memoryBlock(address);
85          end if;
86      end process PortA;
87
88             PortB: process(portB_clk)
89            variable address: integer range 0 to 1023:=0;
90          begin
91             if rising_edge(PortB_clk) then
92                 address:= to_integer(unsigned(portB_addr));
93                if(portB_writeEnable='1') then
94                  memoryBlock(address):=PortB_dataIn;
95                end if;
96                portB_dataOut<= memoryBlock(address);
97             end if;
98        end process PortB;
99
100 end Behavioral;
```

Figure 4-11: Architecture Part for Complete Dual-Port Memory

© Copyright 2019 Xilinx

Lab 5: Using Loops

2019.1

Abstract

This lab demonstrates the usage of loops in a VHDL design. This lab should take approximately 60 minutes.

Objectives

After completing this lab, you will be able to:

- Construct a combination of a generic and generate statement(s) to produce code that will select at elaboration time 8 bits from one of two devices.
- Construct a collection of output buffers using a looping generate statement.
- Build a process in a testbench that serializes data.

© Copyright 2019 Xilinx

Introduction

This design is intended for use in the wave_gen.vhd module and will steer either data from the UART receiver or the sample generator to the LEDs on the development board. You built the multiplexer version of the LED_manager in the "Using Concurrent Statements" lab; however, product specifications have changed and the selection of Channel1 or Channel2 occurs after coding, but before customer use. To save device resources you are being directed to replace the multiplexer with some capability to make this selection during synthesis (without changing the source).

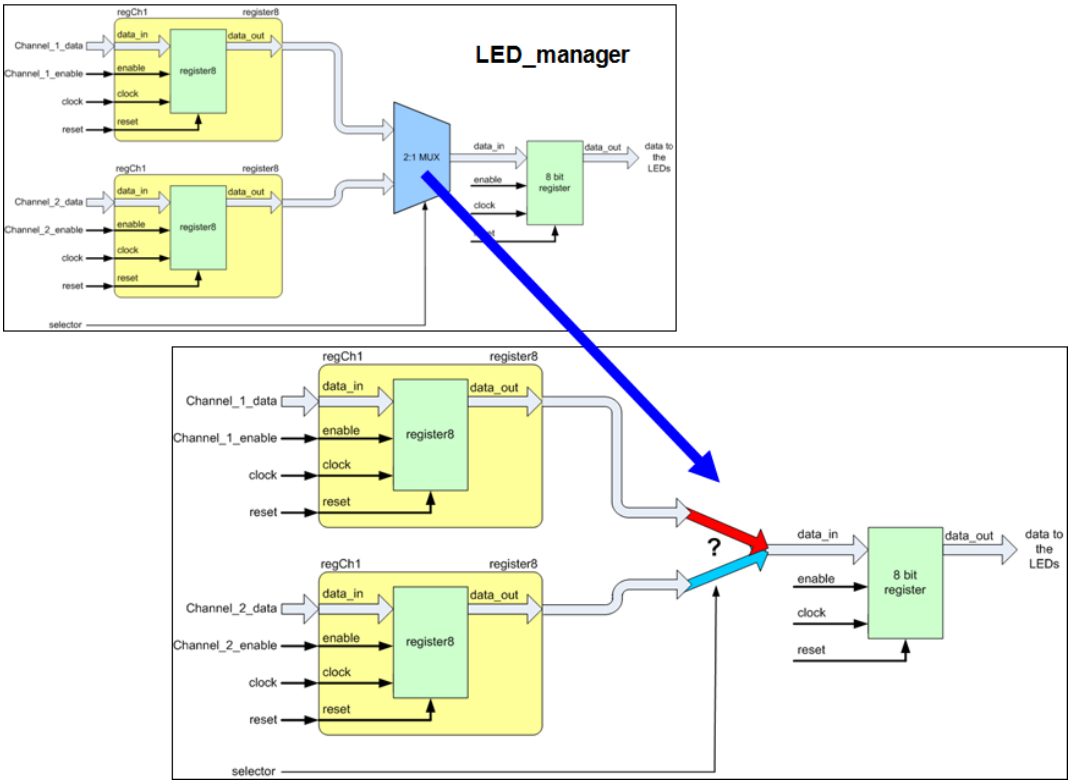


Figure 5-1: Block Diagram for the First Portion of This Lab

You will build upon the "Using Concurrent Statements" lab in which the LED_manager module was constructed for the first portion of the lab. You will modify this module by replacing the 2:1 combinatorial multiplexer with a conditional generate statement.

The second portion of the lab requires that the output buffers for the LED_out signals be instantiated. You will use a looping generate statement to create these buffers.

Finally, you will create a module to replace the serializer that you will use in an upcoming testbench lab.

Understanding the Lab Environment

The labs and demos provided in this course are designed to run under the provided Linux platform. The provided Linux platform has a customized lab environment. These customizations have been made to simplify and enhance your learning experience. Many labs and demos can

© Copyright 2019 Xilinx

be successfully executed in the Windows environment as well. Topics including PetaLinux, Yocto, and QEMU are generally not supported under Windows.

The instructions, for the sake of brevity, are expressed using the Linux notation. This includes the forward slash ('/') as the hierarchy separator instead of the Windows backslash ('\'). For students who want to run under Windows, it is their responsibility to use the correct hierarchy separator.

Windows users can run Linux using the VirtualBox tool from Oracle. Details on how to acquire and install this tool can be found in the lab setup guide. The Xilinx Customer Education group offers a preconfigured virtual machine image and includes all the tools necessary to run the labs. Some tools and IP do require licenses, which are the responsibility of the student.

Several environment variables, described in detail in the lab setup guide, which also discusses how you can customize the environment for your situation, are provided for ease of use. Most notably, many environment variables have been defined to reduce the amount of typing you need to do when entering paths.

These environment variables are:

- XILINX_PATH - points to the installation directory for the Xilinx tools. Tools such as the Vivado Design Suite and SDK can be found here.
- PETALINUX_PATH - points to the installation directory for the Xilinx PetaLinux tools. Often this is within XILINX_PATH, but some choose to install it elsewhere.
- TRAINING_PATH - points to the space allocated for students to work through their labs. This directory includes prebuilt images and starting points for the labs and demos.
- genLoops - points to the specific name of the topic within the TRAINING_PATH directory.

  This lab uses $genLoops to indicate the directory for this lab, which is equivalent to $TRAINING_PATH/genLoops.

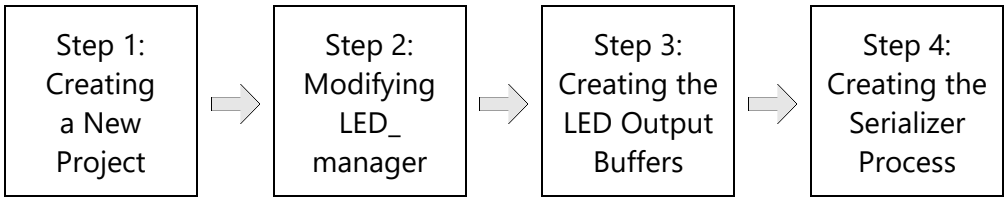
These environment variables work well in the Linux terminal and the Vivado Design Suite and scripts, but not in Eclipse-based tools such as the SDK tool and the SDx™ environment. When using the Vivado tools, you can use $genLoops and the tools will automatically expand the environment variable for you.

However, when you see the notation $genLoops/workspace in SDK or the SDx environment, you should interpret it in its expanded form. The expanded form can be found by opening a terminal window and enter "echo $genLoops".

This lab assumes that you are starting with a Linux environment. If you are working on a Windows system, refer to the lab setup guide to see how to use the Oracle VirtualBox application to run the Xilinx Customer Education pre-configured Ubuntu Linux virtual machine.

© Copyright 2019 Xilinx

General Flow



Creating a New Project — Step 1

1-1. Launch the Vivado Design Suite.

If you do not recall how to perform this task, refer to the "Launching the Vivado Design Suite" section under Vivado Design Suite Operations in the *Lab Reference Guide*.

1-2. Create a new Vivado Design Suite project named *genLoops* and locate it in the following directory.

Browse to the `$genLoops/lab/KCU105` directory.

Target the project for the Kintex UltraScale KCU105 Evaluation Platform evaluation board.

If you do not recall how to perform this task, refer to the "Creating a Blank Vivado Design Suite Project" section in the *Lab Reference Guide*.

From the settings in the Flow Navigator, change the target language to VHDL.

© Copyright 2019 Xilinx

Modifying LED_manager Step 2

In this lab, you will import the LED_manager design file that was used in the "Using Concurrent Statements" lab. This design has a multiplexer which selects between two channels. This multiplexer is replaced by a conditional generated here. When this is complete, you will synthesize the design and compare the device resources with the original device resources as well as reviewing the RTL code.

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

Now you will add the LED_manager and register8 modules (provided in the current working directory) into the project.

2-1. Add an HDL source file to the design.

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

The Add Sources Wizard opens.

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

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

2-1-4. Click the **Plus** (+) icon and select **Add Files**.

2-1-5. Browse to the `$genLoops/lab/KCU105` directory if it is not open already.

2-1-6. Select **LED_manager.vhd and register8.vhd**.

2-1-7. Click **OK**.

2-1-8. Ensure that the **Copy sources into project** option is selected.

2-1-9. Click **Finish** in the Add or Create Design Sources dialog box to add the HDL sources to the project.

2-2. Examine the multiplexer.

2-2-1. Select the **Sources** view.

2-2-2. Double-click **LED_manager** to open the source code in the text editor.

2-2-3. Examine the code.

2-3. Run synthesis.

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

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

© Copyright 2019 Xilinx

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

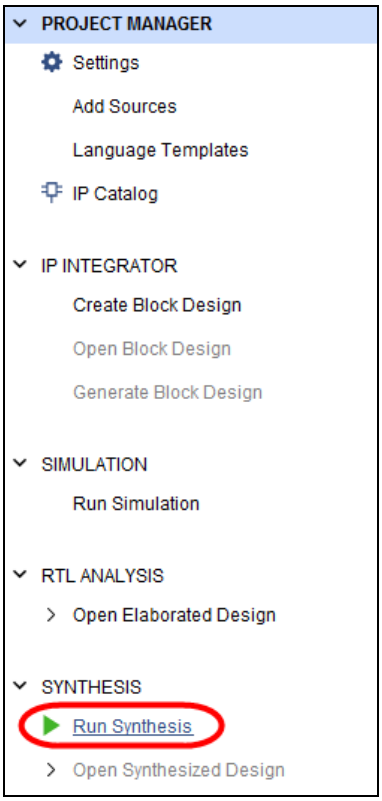


Figure 5-2: Selecting Run Synthesis

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

After the synthesis process completes, the Synthesis Completed dialog box opens. The dialog box prompts you to run implementation, open the synthesized design, or view reports.

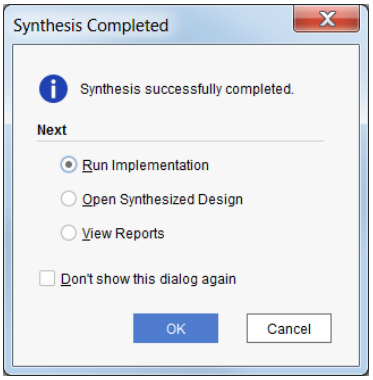


Figure 5-3: Synthesis Completed Dialog Box

2-3-4. Click **OK** to continue with your preferred choice or **Cancel** to simply close the dialog box and return to the normal view of the Vivado Design Suite.

© Copyright 2019 Xilinx

2-4. Generate a Utilization report.

2-4-1. Click **Report Utilization** under Synthesized Design in the Flow Navigator.

Alternatively, you can select **Reports** > **Report Utilization**.

Make sure that the synthesized design is open at this point in order to access the reports.

2-4-2. Click **OK** for the default name utilization_1 in the pop-up window.

Question 1

Complete the Multiplexer Version column in the table below.

	Multiplexer Version	**Other Version**
Number of Slice Registers		
Number of Slice LUTs		

Resources Table

Question 2

How will you code the multiplexer so that its selection will be fixed at synthesis?

_____

_____

_____

Question 3

Identify the inputs for the multiplexer. Which signals will and will not be used to implement the conditional generate?

_____

_____

_____

2-4-3. Close the synthesized design.

© Copyright 2019 Xilinx

2-5. In LED_manager.vhd, replace the multiplexer with a conditional generate. Create a generic named CHAN_SEL of an integer type with a range from 1 to 2 to use as the selector.

Although suggested code solutions can be found in the Answers section, you should make every effort to code these yourself.

2-5-1. Highlight the code and click the **Comment** icon // in the toolbar of the text editor to comment out the multiplexer.

```
  --
  -- construct a 2 input mux
-- with Selector select
--    mux_data_selected <= Channel_1_registered_data when '0',
--                         Channel_2_registered_data when '1',
--                         (others=>'-')             when others;
```

Figure 5-4: Commenting out the Multiplexer

2-5-2. Create the conditional generate to use the generic CHAN_SEL (which will be defined in a couple of steps) to drive Channel_1_registered_data to the mux_data_selected signal.

```
c1Sel: if (CHAN_SEL = 1) generate
            mux_data_selected <= Channel_1_registered_data;
       end generate c1Sel;
```

Figure 5-5: Conditional generate to drive Channel_1_registered_data

2-5-3. Repeat to create a second conditional generate to drive Channel_2_registered_data to the mux_data_selected signal.

2-5-4. Add a generic to the entity named CHAN_SEL which is an integer that can have the value of either 1 or 2 (to serve as the selection mechanism between channels 1 and 2).

2-5-5. Select **File** > **Text Editor** > **Save File**.

2-6. Synthesize LED_manager and open the Design Summary.

2-6-1. Click **Run Synthesis** under Synthesis in the Flow Navigator.

2-6-2. Select the **Open Synthesized Design** option in the dialog box and click **OK** if the Synthesis Completed dialog box appears.

2-7. Generate a Utilization report.

2-7-1. Click **Report Utilization** under Synthesized Design in the Flow Navigator.

Alternatively, you can select **Reports** > **Report Utilization**.

2-7-2. Click **OK** for the default name **utilization_1** in the pop-up window.

© Copyright 2019 Xilinx

Question 4

Complete the Other Version column in the Resources table.

	Multiplexer Version	Other Version
Number of Slice Registers		
Number of Slice LUTs		

Resources Table

Question 5

Compare the results between the two synthesis runs. Explain the differences.

_____

_____

_____

2-7-3. Close the synthesized design.

© Copyright 2019 Xilinx

Creating the LED Output Buffers — Step 3

The synthesis tools are capable of recognizing the top-level signals in a design and placing basic I/O buffers on them. However, sometimes you may want to take control from the synthesis tools and define buffers manually. This is typically done when the I/O buffers need special qualities associated with them, such as double-data rate or differential signaling, for example.

You will instantiate the output buffers for the LEDs in the UART_led module in this step.

3-1. Add the UART_led to the design and its supporting files:

- **uart_led.vhd**
- **meta_harden.vhd**
- **reset_bridge.vhd**
- **uart_rx.vhd**
- **uart_rx_ctl.vhd**
- **uart_baud_gen.vhd**

3-1-1. Click **Add Sources** under Project Manager in the Flow Navigator.

The Add Sources Wizard opens.

3-1-2. Select **Add or create design sources**.

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

3-1-4. Click the **Plus** (+) icon and select **Add Files**.

3-1-5. Press the <**Ctrl**> key and select all the files listed above.

3-1-6. Click **OK** and **Finish** to add the selected files to the project.

© Copyright 2019 Xilinx

3-2. Disable the synthesis tool's ability to automatically insert I/O buffers.

3-2-1. Select **Settings** in the Flow Navigator.

Select **Synthesis** in the settings dialog box.

3-2-2. Enter **-no_iobuf** in the More Options field.

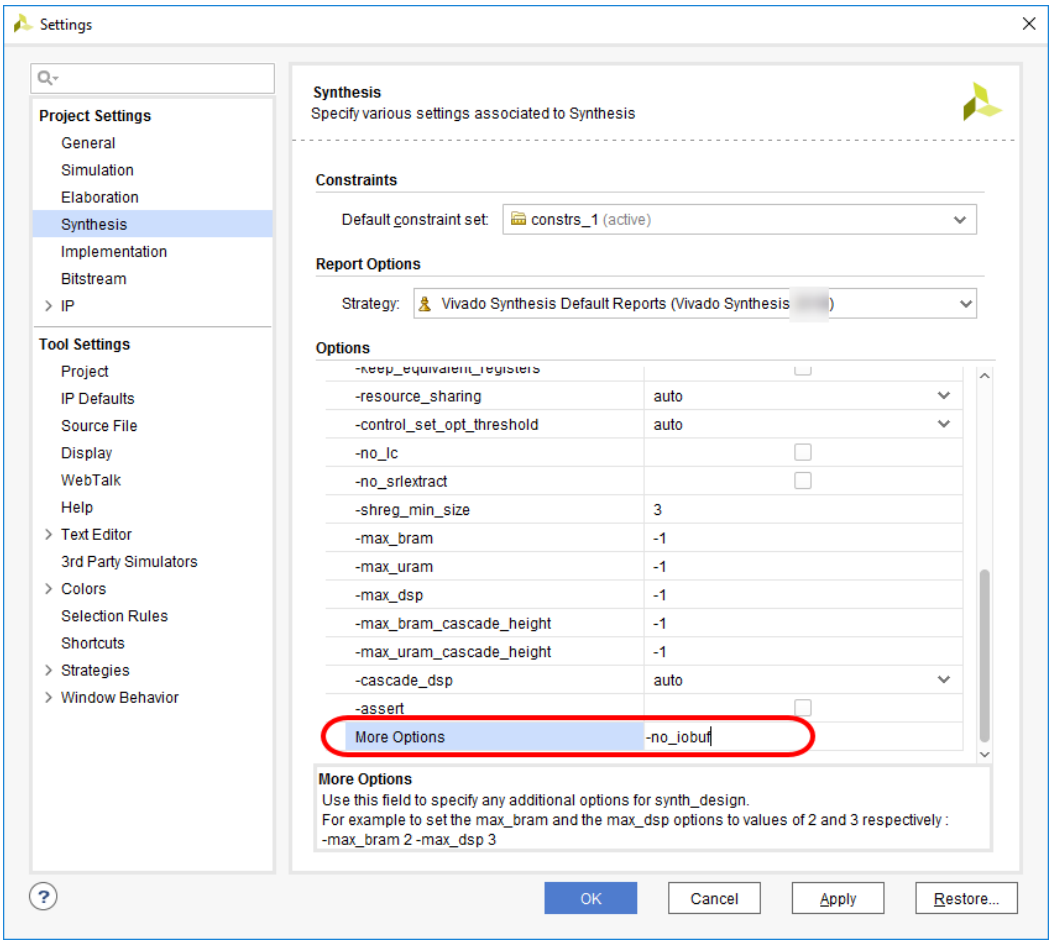


Figure 5-6: Disabling Automatic I/O Buffer Insertion

3-2-3. Click **OK**.

3-3. Add the appropriate generate statement to create eight instances of the OBUF, one for each LED output.

The OBUF is a primitive that is defined in Xilinx's UNISIM library in the VComponents package.

Although suggested code solutions can be found in the Answers section, you should make every effort to code these yourself.

© Copyright 2019 Xilinx

Question 6

How can you determine what primitive component to use and how it should be connected?

_____

_____

_____

3-3-1. Select the **Sources** view.

3-3-2. Double-click the **uart_led** module to open the file in the text editor.

3-3-3. Select **Tools** > **Language Templates** to open the Language Templates.

1. Expand **VHDL** > **Device Primitive Instantiation** > **Kintex UltraScale** > **I/O** > **OUTPUT_BUFFER**.
2. Select **Output Buffer (OBUF)**.

3-3-4. Add the instantiation template to uart_led.vhd at the location indicated by the comment "—define the buffers for the outgoing data".

3-3-5. Wrap the instantiation with a for-loop generate statement.

3-3-6. Change the **OBUF_inst** name to **OBUF_led_i**.

3-3-7. Change the input signal from 'I' to the output of LED_manager (LED_o).

Remember that this signal is a bus and that OBUF takes a single value.

Hint: You will have to represent the bus as a series of slices.

3-3-8. Change the output signal from 'O' to the pin assignment *led_pins*.

As with the previous step you will have to represent this as a slice.

3-3-9. Select **File** > **Text Editor** > **Save File**.

Question 7

What library needs to be loaded to use primitives? Why was this not a step in this process?

_____

_____

© Copyright 2019 Xilinx

3-4. Now that the synthesis options have been set, launch synthesis and verify that the output buffers were properly integrated into the design.

3-4-1. Click **Run Synthesis** under Synthesis in the Flow Navigator.

3-4-2. After synthesis finishes, select **Open Synthesized Design** in the Synthesis Completed dialog box and click **OK**.

3-4-3. Select the **Netlist** view.

3-4-4. Right-click **uart_led** and select **Schematic** to launch the schematic viewer.

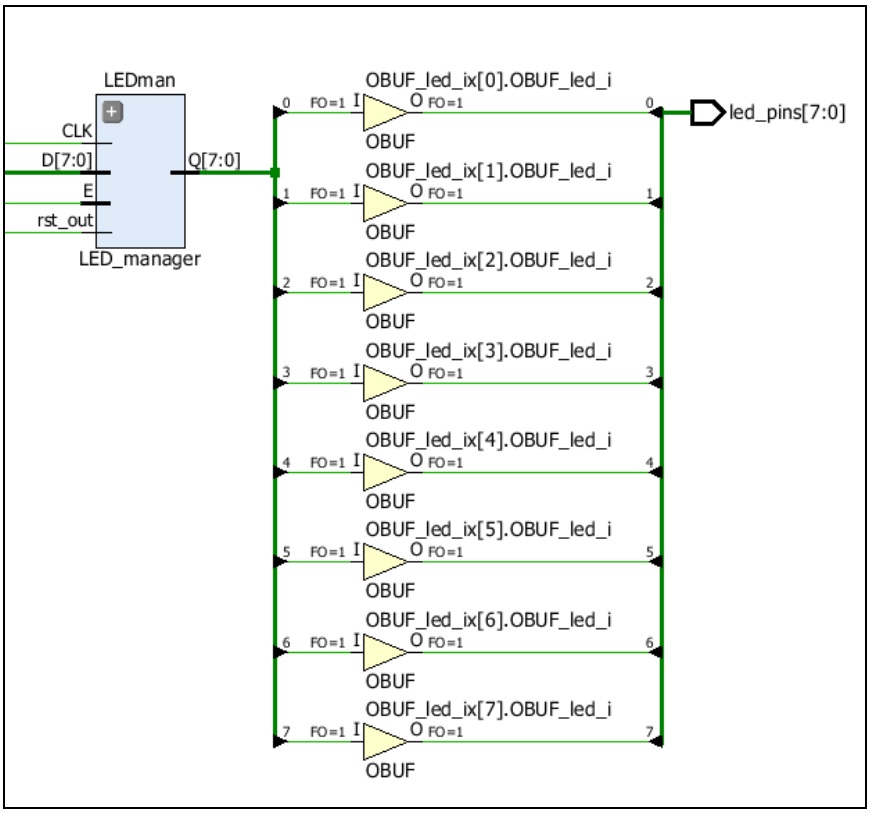


Figure 5-7: Schematic Showing the Eight OBUFs

3-4-5. Close the viewer windows.

© Copyright 2019 Xilinx

Creating the Serializer Process Step 4

The serializer process takes a provided string, breaks the string into characters, and converts the characters into a binary serial usable for testing an RS-232 receiver. The parameters for this process are 115,200 baud, 8 data bits, 1 stop bit, no parity, and no flow control.

The outline of the module has been provided for you. You are only required to fill in part of the process.

For your reference, here is a generic RS-232 frame.

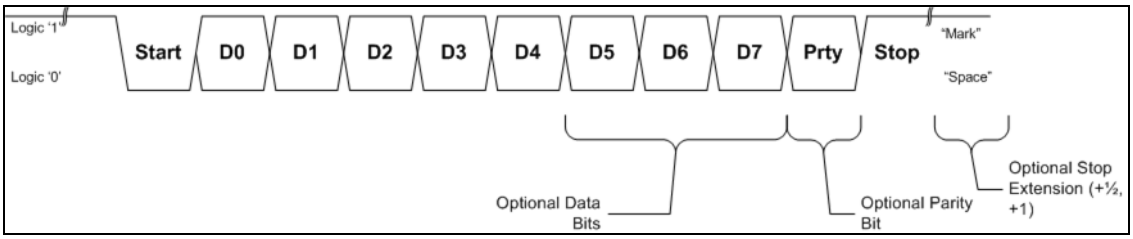


Figure 5-8: Generic RS-232 Frame

4-1. Load tb_uart_driver.vhd for simulation only.

This is a "helper" module for the final testbench that you will assemble in the last lab of this course.

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

The Add Sources Wizard opens.

4-1-2. Select **Add or create simulation sources**.

4-1-3. Click **Next**.

4-1-4. Click the **Plus** (+) icon and select **Add Files**.

4-1-5. Select **tb_uart_driver.vhd** and click **OK**.

This will keep the tb_uart_driver associated only with simulation and will not appear in the implementation sources.

4-1-6. Click **Finish** to close the Add and Create Simulation Sources dialog box.

© Copyright 2019 Xilinx

4-2. Set the tb_uart_driver.vhd as the top-level file for simulation.

4-2-1. Expand **Simulation Sources** in the Sources view.

4-2-2. Right-click **tb_uart_driver.vhd** under **Simulation Sources** > **sim_1** and select **Set as Top**.

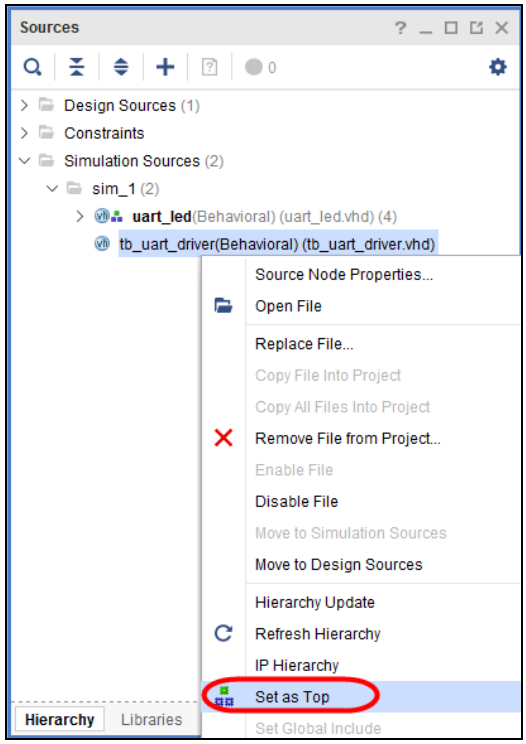


Figure 5-9: Selecting tb_uart_driver.vhd as the Top-Level File

4-3. Open tb_uart_driver and write the code to generate RS-232 style packets based on the provided "MESSAGE".

Although suggested code solutions can be found in the Answers section, you should make every effort to code these yourself.

4-3-1. Select the **Sources** view.

4-3-2. Expand **Simulation Sources** and double-click **tb_uart_driver.vhd**.

4-3-3. Locate the comment "-- loop through the entire message" in the source code.

4-3-4. Write the code to break the string described in the generic "MESSAGE" into characters.

Hint: Try using a for loop.

4-3-5. Write the code to convert the character into an 8-bit std_logic_vector named slv_char_to_send.

Hint: The attribute ‘pos can be used in the synthesis environment to return the ASCII equivalent of a character. The use in this case is character’pos(<character you want to send>).

© Copyright 2019 Xilinx

4-3-6. Write the code to send the start bit.

Hint: The start bit is a logic 0 that is presented to the serial line for one baud period.

Hint: There is a pre-defined constant that will help you.

4-3-7. Write the code to send all 8 bits of the character to send with the most significant bit sent last.

Hint: Use a nested for loop. Be sure not to name this loop identifier the same as the outer loop identifier.

4-3-8. Write the code to generate a 1-bit period high signal to indicate the stop bit.

Hint: This is exactly like the start bit, but with a different value.

4-3-9. Write the code to create a 10-ns pulse to indicate that the character was sent

4-3-10. Write the code to wait 5 bit periods before sending the next packet.

4-3-11. Select **File** > **Text Editor** > **Save File**.

4-4. Simulate the design. Load the provided configuration file genLoops.wcfg. Set the run time for 2 ms and examine the waveform.

4-4-1. Select **Simulation** > **Run Simulation** > **Run Behavioral Simulation** in the Sources view.

This will launch the Vivado simulator.

4-4-2. Select **File** > **Simulation Waveform** > **Open Configuration**.

4-4-3. Select **genLoops.wcfg** and click **OK**.

4-4-4. Select **Run** > **Restart** (or click the icon) to restart the simulation.

4-4-5. Select **Run** > **Run All** (or click the icon) to run the simulation.

4-4-6. Click the **Zoom to Fit** ( ) icon to zoom to full screen.

Question 8

Is the design working? How can you tell?

_____

_____

4-5. Close the simulation.

4-5-1. Select **File** > **Close Simulation**.

4-5-2. Click **OK** in the Confirm Close window.

4-5-3. Click **Discard** in the pop-up dialog box.

© Copyright 2019 Xilinx

4-6. Close the Vivado Design Suite.

4-6-1. Select **File** > **Exit**.

The Exit Vivado dialog box opens.

Figure 5-10: Exit Vivado Dialog Box

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

[Optional]: Clean up the file system.

Some systems 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. Both processes will recursively delete all of the files in the `$genLoops` directory.

4-7. Clean up the VirtualBox file system via the GUI.

4-7-1. Using the graphical browser (Windows: press the <**Windows**> key + <**E**>; Linux: press <**Ctrl** + **N**>), navigate to `$genLoops`.

4-7-2. Select **`genLoops`**.

4-7-3. Press <**Delete**>.

4-8. Clean up the VirtualBox file system via the command line.

4-8-1. Open a terminal window (Windows: press the <**Windows**> key + <**R**>, then enter **cmd**; Linux: press <**Ctrl** + **Alt** + **T**>)

4-8-2. Enter the following command to delete the contents of the workspace:

[Windows users]: **`rd /s /q $genLoops`**

[Linux users]: **`rm -rf $genLoops`**

© Copyright 2019 Xilinx

Summary

This lab walked you through three common activities when coding:

- Making a resource versus functionality trade-off while still preserving some amount of flexibility
- Using generate loops to reduce the time it takes to code redundant activities (and to keep the code more concise)
- Using loop statements to ease the repetition when creating stimulus

© Copyright 2019 Xilinx

Answers

Answers listed represent sample solutions only. Your results may differ depending on the version of the software, service pack, or operating system that you are using.

1. Complete the Multiplexer Version column in the table below.

	Multiplexer Version	**Other Version**
Number of Slice Registers	24	
Number of Slice LUTs	4	

Resources Table

2. How will you code the multiplexer so that its selection will be fixed at synthesis?

   A conditional generate based on a generic is one possibility for solving this problem.

3. Identify the inputs for the multiplexer. Which signals will and will not be used to implement the conditional generate?

   The multiplexer takes Channel_1_registered_data and Channel_2_registered_data as inputs. This will not change. The output remains mux_data_selected. The remaining signal "Selector" is currently a signal which can change during the operation of the design which will cause mux_data_selected to dynamically switch input channels. Because the selector will be fixed at synthesis, this selector signal will go unused.

   This implies the need for some mechanism to select which channel will be enabled during synthesis. Use a generic to make this selection.

4. Complete the Other Version column in the Resources table.

	Multiplexer Version	**Other Version**
Number of Slice Registers	24	16
Number of Slice LUTs	4	0

Resources Table

5. Compare the results between the two synthesis runs. Explain the differences.

   The conditional generate version of the code used fewer resources than the multiplexer version. This is not surprising as all of the code for the multiplexer itself (which was implemented as LUTs) no longer needs to be built. The implementation becomes only

© Copyright 2019 Xilinx

connections. Because only one channel needs to be registered, the eight flip-flops used to register the other channel have been optimized away.

This is a clear example between choosing flexibility or minimal resources.

6. How can you determine what primitive component to use and how it should be connected?

   One method is to refer to document known as the *<FPGA family name> Libraries Guide for HDL Designs*. These documents, one per FPGA family, describe the behavior of all of the primitive components and their interface.

   Here is a small portion of the manual for the OBUF.

OBUF

Primitive: Output Buffer

Introduction

This design element is a simple output buffer used to drive output signals to the FPGA device pins that do not need to be 3-stated (constantly driven). Either an OBUF, OBUFT, OBUFDS, or OBUFTDS must be connected to every output port in the design.

This element isolates the internal circuit and provides drive current for signals leaving a chip. It exists in input/output blocks (IOB). Its output (O) is connected to an OPAD or an IOPAD. The interface standard used by this element is LVTTL. Also, this element has selectable drive and slew rates using the DRIVE and SLOW or FAST constraints. The defaults are DRIVE=12 mA and SLOW slew.

Port Descriptions

Name	Direction	Width	Function
O	Output	1-bit	Output of OBUF to be connected directly to top-level output port.
I	Input	1-bit	Input of OBUF. Connect to the logic driving the output port.

Instantiation	Yes
Inference	Recommended
Coregen and wizards	No
Macro support	No

Figure 5-11: Snippet from the Spartan-3E FPGA Libraries Guide for HDL Designs

Another quicker method is to use the Language Templates. The templates provide the instantiation template that can be cut-and-pasted directly into your code (however, there is little or no description about how the component works).

Another advantage to the templates is that the primitives are sorted by function, not by name as they are in the Libraries Guides. This makes finding the right component easier if you do not know the exact name.

© Copyright 2019 Xilinx

7. What library needs to be loaded to use primitives? Why was this not a step in this process?

   The VComponents package within the UNISIM library contains the declarations for all of the primitive components and associated macros. This library was already loaded to support the other input and output buffers.

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

   The design seems to be working.

   The first item to test is the baud rate: measure from one data_sent pulse to the next. This can be done using the click-and-drag markers or placing two cursors and computing the delta between them.

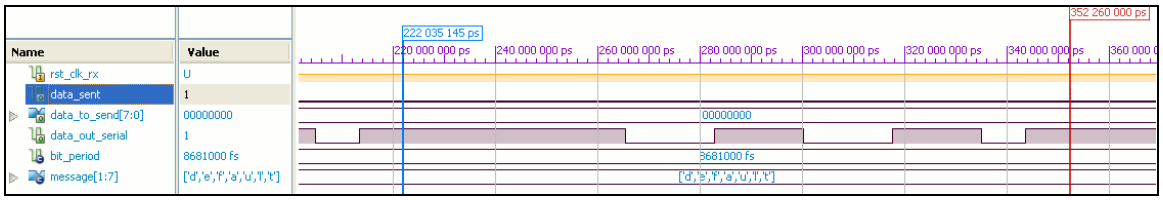


Figure 5-12: Measuring Baud Rate

The difference between the two data_sent pulses is 130.225 µs. This includes 5 bit periods for the delay between frames and the start, stop, and data bits for a total of 15 bit periods or 8.68 µs per pulse which results in 115,200 baud. So, this seems to be correct.

Next, you can pick one of the frames and manually decode it. Because you measured the third frame in time, you might as well decode it.

Notice the one shorter bit just prior to the stop bit? Start there and work backwards.

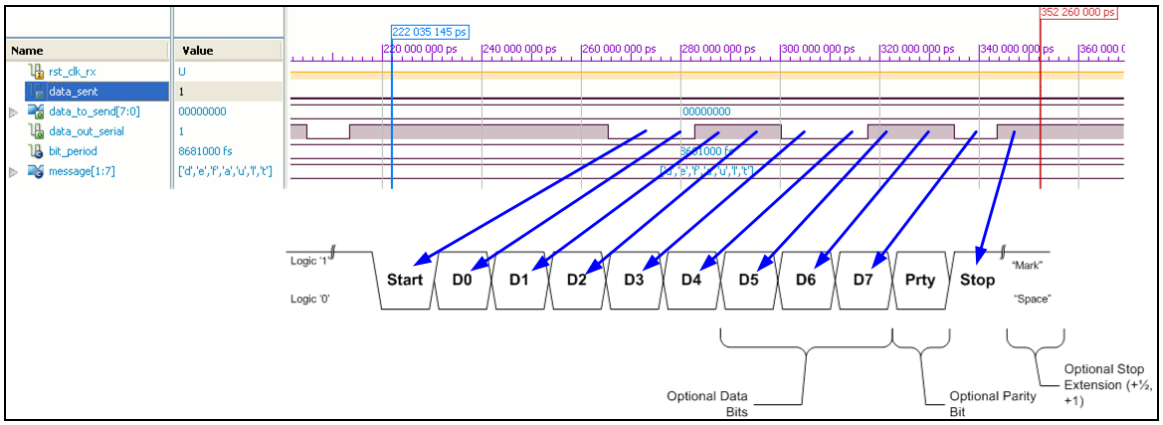


Figure 5-13: Decoding the Waveform

According to this, the result is hexadecimal 66, which is the character code for ‘f’. This, not surprisingly, is the third character sent in the message.

Therefore, you can conclude that this design is working correctly.

© Copyright 2019 Xilinx

Final Code

Code to create instances of the OBUF

```
-- define the buffers for the outgoing data
 OBUF_led_ix : for i in 0 to 7 generate
   OBUF_led_i: OBUF
       generic map (
           DRIVE => 12,
           IOSTANDARD => "DEFAULT",
           SLEW => "SLOW")
       port map (
           O => led_pins(i),    -- Buffer output (connect directly to top-level port)
           I => LED_o(i)          -- Buffer input
       );
end generate OBUF_led_ix;
```

Figure 5-14: Creating Instances of the OBUF

The final code of tb_uart_driver.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
library std;
use STD.TEXTIO.all;
entity tb_uart_driver is
    generic (
        MESSAGE        : String := "default";
        BIT_PERIOD    : time := 8681 ns
      );
   port (
        rst_clk_rx            : in    std_logic;
        data_sent               : out std_logic;
        data_to_send          : out std_logic_vector(7 downto 0);
        data_out_serial     : out std_logic
      );
end tb_uart_driver;
architecture Behavioral of tb_uart_driver is
```


© Copyright 2019 Xilinx

```
    begin
        send_char_bitper: process
            variable char_to_send          : character := '?';
            variable char_code             : integer range 0 to 255 := 0;
            variable slv_char_to_send      : std_logic_vector(7 downto 0) := (others=>'U');
            variable char_as_string        : string (1 to 1);
            variable hex_as_string         : string (1 to 6);
            variable console_message       : string (1 to 40);
          begin
            data_to_send          <= (others=>'0');
            data_out_serial       <= '1';
            data_sent             <= '0';
--            wait until (rst_clk_rx = '0');
            wait for 5 us;
            -- loop through the entire message
             sendChars: for i in 1 to MESSAGE'length loop
               -- break the string into characters
               -- convert the character into a std_logic_vector of 8 bit
              char_to_send        := MESSAGE(i);    -- convert char to slv8
              char_code           := character'pos(char_to_send);    -- char to int
              slv_char_to_send := std_logic_vector(to_unsigned(char_code,8));
               -- send the start bit, then the 8 data bits, then the stop bit
               data_out_serial      <= '0';              -- send start bit
               wait for BIT_PERIOD;                      -- and hold for one bit period
             -- loop through the 8 bits of data and send lsb first
               for j in 0 to 7 loop                      -- loop through all 8 bits
                    data_out_serial <= slv_char_to_send(j);
                    wait for BIT_PERIOD;                                    -- and hold for one
bit period
               end loop;
               data_out_serial <= '1';                                -- send stop bit
               wait for BIT_PERIOD;                                    -- delay a period
```


© Copyright 2019 Xilinx

```
            data_sent      <= '1';
            wait for 10 ns;
            data_sent <= '0';
            wait for BIT_PERIOD*5;                                  -- idle 5 bit periods
        end loop; -- sendChars
        wait for 1 ms;
        assert false
        report "***** Simulation Ending due to all data being transmitted! *****"
        severity failure;
        wait;
    end process send_char_bitper;
end Behavioral;
```


© Copyright 2019 Xilinx

Lab 6: Simulating a Simple Design

2019.1

Abstract

This lab emphasizes the flexibility of processes within a testbench. It should take approximately 45 minutes.

Objectives

After completing this lab, you will be able to:

- Demonstrate the use of processes to generate clock and reset stimulus
- Use the Vivado® simulator to navigate the output waveform

Introduction

During this lab, you will create processes that generate a 50/50 duty cycle clock and a reset that always deasserts after 25 clock pulses regardless of the clock period. A data generation module is provided and must be instantiated within the testbench.

You will be guided through some of the more advanced features of the Vivado simulator.

Understanding the Lab Environment

The labs and demos provided in this course are designed to run under the provided Linux platform. The provided Linux platform has a customized lab environment. These customizations have been made to simplify and enhance your learning experience. Many labs and demos can be successfully executed in the Windows environment as well. Topics including PetaLinux, Yocto, and QEMU are generally not supported under Windows.

The instructions, for the sake of brevity, are expressed using the Linux notation. This includes the forward slash ('/') as the hierarchy separator instead of the Windows backslash ('\'). For students who want to run under Windows, it is their responsibility to use the correct hierarchy separator.

Windows users can run Linux using the VirtualBox tool from Oracle. Details on how to acquire and install this tool can be found in the lab setup guide. The Xilinx Customer Education group offers a preconfigured virtual machine image and includes all the tools necessary to run the labs. Some tools and IP do require licenses, which are the responsibility of the student.

Several environment variables, described in detail in the lab setup guide, which also discusses how you can customize the environment for your situation, are provided for ease of use. Most notably, many environment variables have been defined to reduce the amount of typing you need to do when entering paths.

© Copyright 2019 Xilinx

These environment variables are:

- XILINX_PATH - points to the installation directory for the Xilinx tools. Tools such as the Vivado Design Suite and SDK can be found here.
- PETALINUX_PATH - points to the installation directory for the Xilinx PetaLinux tools. Often this is within XILINX_PATH, but some choose to install it elsewhere.
- TRAINING_PATH - points to the space allocated for students to work through their labs. This directory includes prebuilt images and starting points for the labs and demos.
- basicTestbench - points to the specific name of the topic within the TRAINING_PATH directory.

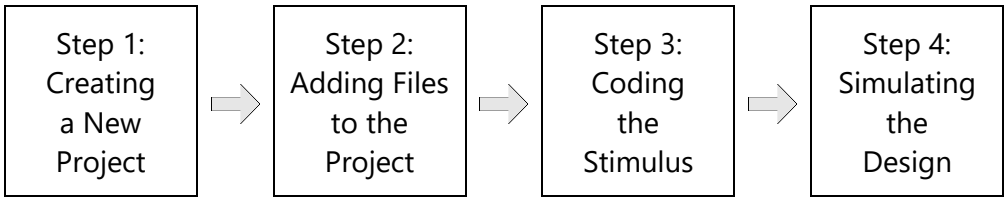
  This lab uses $basicTestbench to indicate the directory for this lab, which is equivalent to $TRAINING_PATH/basicTestbench.

These environment variables work well in the Linux terminal and the Vivado Design Suite and scripts, but not in Eclipse-based tools such as the SDK tool and the SDx™ environment. When using the Vivado tools, you can use $basicTestbench and the tools will automatically expand the environment variable for you.

However, when you see the notation $basicTestbench/workspace in SDK or the SDx environment, you should interpret it in its expanded form. The expanded form can be found by opening a terminal window and enter "echo $basicTestbench".

This lab assumes that you are starting with a Linux environment. If you are working on a Windows system, refer to the lab setup guide to see how to use the Oracle VirtualBox application to run the Xilinx Customer Education pre-configured Ubuntu Linux virtual machine.

General Flow



© Copyright 2019 Xilinx

Creating a New Project Step 1

1-1. Launch the Vivado Design Suite.

If you do not recall how to perform this task, refer to the "Launching the Vivado Design Suite" section under Vivado Design Suite Operations in the *Lab Reference Guide*.

1-2. Create a new Vivado Design Suite project named *basicTestbench* and locate it in the following directory.

Browse to the `$basicTestbench/lab/KCU105` directory.

Target the project for the Kintex- UltraScale KCU105 Evaluation Platform evaluation board.

If you do not recall how to perform this task, refer to the "Creating a Blank Vivado Design Suite Project" section in the *Lab Reference Guide*.

From the settings in the Flow Navigator, change the target language to VHDL.

In this lab, you will not generate the bitstream and verify the design on hardware; it does not require the KCU105 board for you to download the bitstream, just the project you will be targeting to the KCU105 board.

© Copyright 2019 Xilinx

Adding Files to the Project — Step 2

The uart_led design is a sub-set of the larger wave_gen design which is the common design used in the many of Xilinx's courses.

The source files for the uart_led have been preloaded into the working directory and you will be using these design files to perform simulation.

2-1. Add the following design files into the project.

- **LED_manager – multiplexer to select functionality**
- **meta_harden – anti-meta-stable circuit**
- **register8.vhd – 8-bit registering flip-flop module**
- **reset_bridge – for safely applying reset**
- **uart_baud_gen – baud rate generator for oversampling serial data**
- **uart_driver_tb – creates stimulus for the testbench**
- **uart_led.vhd – top level of the design**
- **uart_led_simple_tb – simplified testbench**
- **uart_rx.vhd – UART receiver**
- **uart_rx_ctl – UART state machine**

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

The Add Sources Wizard opens.

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

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

2-1-4. Click the **Plus** (+) icon and select **Add Files**.

2-1-5. Press the <**Ctrl**> key and select all the files listed above.

2-1-6. Click **OK**.

2-1-7. Click **Finish** to confirm adding the selected files to the project with the default associations and libraries.

The files that are added can now be seen in the Sources view. The uart_led_simple_tb and tb_uart_driver files are simulation only files and should be moved to the simulation only category.

© Copyright 2019 Xilinx

2-2. Associate the testbench files to simulation from the Sources view.

2-2-1. Right-click **uart_led_simple_tb** under Design Sources and select **Move to Simulation Sources.**

The file is now moved under Simulation Sources in the Sources view.

2-2-2. Repeat this step for **tb_uart_driver** to be moved under Simulation sources.

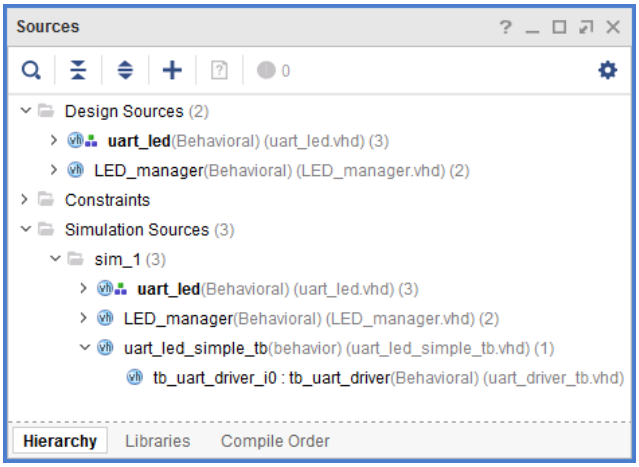


Figure 6-1: Sources Window

You will also see some of the other modules that are instantiated in uart_led_simple_tb are also seen in the simulation hierarchy.

© Copyright 2019 Xilinx

2-3. Open the top file of the design, examine the contents, and complete the code to make necessary connections as shown in the figure below.

uart_led.vhd is the top-level source file for the uart_led project and will contain the completed code for synthesis. All that remains is to properly connect LED_manager.

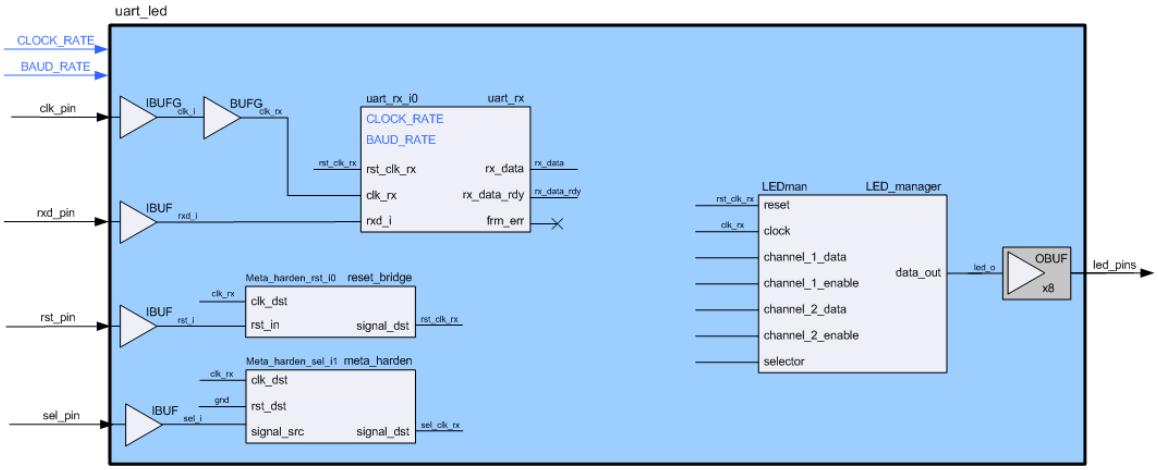


Figure 6-2: Block Diagram of uart_led design

2-3-1. Double-click **uart_led** in the Hierarchy view to open the source file.

2-3-2. Uncomment the LED_manager instantiation at lines 191 to 200.

2-4. Output of the UART receiver must be connected to the LED_manager.

Make the appropriate signal connections to the LED_manager instantiation in uart_led and save the file.

2-4-1. Connect **rx_data** to Channel_1_data and **rx_data_rdy** to Channel_1_enable.

Connect the meta-hardened selection line **sel_i** from meta_harden_sel_i1 to the selector input on the LED_manager.

The clock input to the LED_manager is driven by **clk_rx**.

Not surprisingly, the reset of LED_manager is connected to **rst_clk_rx**.

The data_out from the LED_manager should tie to the inputs of the OBUFs using **LED_o**. This leaves the inputs to Channel_2 open.

2-4-2. Tie these signals to ground using the constant named **gnd**.

Note that the data is the full 8 bits, but the enable is only 1 bit. Use a slice of gnd (i.e., **gnd(0)**) for the enable.

2-4-3. Select **File** > **Text Editor** > **Save File**.

© Copyright 2019 Xilinx

2-5. Synthesize the design with the default options.

2-5-1. Click **Run Synthesis** in the Flow Navigator and click **OK.**

2-5-2. Correct any errors if found and resynthesize.

2-5-3. Click **Cancel** if the Synthesis Completed dialog box appears.

Coding the Stimulus Step 3

In this step, you will step through the creation of the stimulus for uart_led design. The data generation module, which uses a number of techniques that will be introduced later in this class, has been provided for you.

Your will first create a clock which produces a periodic 50% duty cycle square wave running at 100 MHz. You will then build a mechanism that holds reset asserted until 100 clock cycles have passed. Next, you will build a selector to switch between the grounded Channel 2 and the received data. Finally, you will connect all the blocks of code together and check syntax.

3-1. Using the coding style of your choice, create a block of code in the uart_led_simple_tb.vhd file to create a 100-MHz clock oscillator with a 50% duty cycle.

Although a suggested code solution can be found in the Answers section, you should make every effort to code this yourself.

3-1-1. Expand **Simulation Sources** in the Sources view and double-click **uart_led_simple_tb.vhd** to open the testbench source file.

3-1-2. Write a process or concurrent statement that generates a 100-MHz clock below the line of code which reads "-- generate the clk_rx signal" on or near line 144.

Hint: Use the CLK_RX_PERIOD constant to compute the half period.

3-2. The next block of code that needs to be written is that of the reset.

Code the reset so that it is active high for the first 100 clock periods of the simulation then drops low.

Although a suggested code solution can be found in the Answers section, you should make every effort to code this yourself.

3-2-1. Write a process or concurrent statement that generates an active high reset for 100 clock pulses below the line of code which reads "-- generate the reset signal".

Hint: Use the CLK_RX_PERIOD constant to delay the deassertion of reset.

© Copyright 2019 Xilinx

3-3. The next task is to create a selector stimulus. Because the serial data being pumped out of tb_uart_driver is at 115200 baud, it can safely be assumed that one "frame" of data will be sent in approximately 100 µs. There are 45 characters to send.

Therefore, select Channel 1 and hold for 4.5 ms, then switch to Channel 2. If everything works, this will allow the message to be presented, then zeros after 4.5 ms.

Although a suggested code solution can be found in the Answers section, you should make every effort to code this yourself.

3-3-1. Write a process or concurrent statement that holds select low for 4.5 ms and then switches to Channel 2 (select high) below the line of code which reads "-- generate the select signal".

3-4. Because this testbench was modified from an earlier design, verify that the connections to tb_uart_driver are correct.

Although a suggested code solution can be found in the Answers section, you should make every effort to code this yourself.

3-4-1. Locate tb_uart_driver and verify that all the input pins are being driven. If they are not, connect them to the proper sources.

3-5. The last portion of preparing the testbench for use is to connect the UUT to the various stimuli.

Connect the generated stimulus from the clock, reset, and data generator to the UUT.

Although a suggested code solution can be found in the Answers section, you should make every effort to code this yourself.

3-5-1. Uncomment the lines after the comment "-- instantiate the unit under test" near to the end of the instantiation.

3-5-2. Connect the signals to the appropriate stimulus or sink.

3-5-3. Save the file.

© Copyright 2019 Xilinx

Simulating the Design Step 4

In the final portion of this lab, you will run the simulation and view the waveform.

4-1. Set the default run time for the Vivado simulator to 4.5 ms and launch the Vivado simulator.

4-1-1. Set the **uart_led_simple_tb.vhd** as top level file by right clicking and selecting **Set as Top**.

4-1-2. Select **Settings** in the Flow Navigator.

4-1-3. Select the **Simulation** view in the Project Settings dialog box and select the **Simulation** tab.

4-1-4. Change the Simulation Run Time default value from 1000 ns to **4.5 ms**.

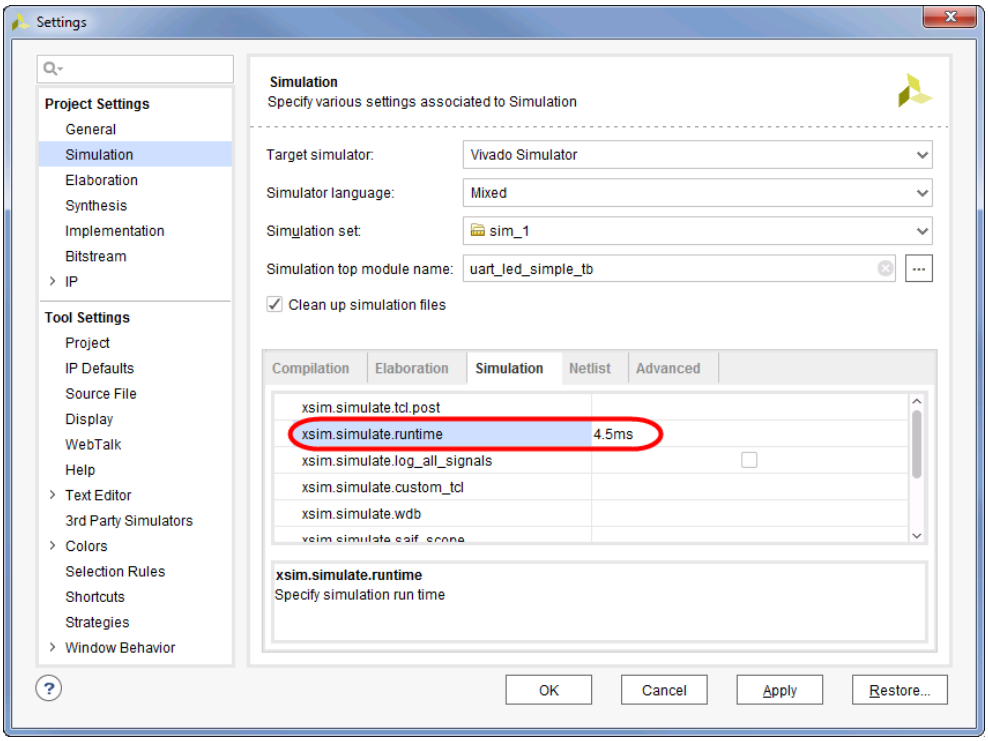


Figure 6-3: Setting New Default Simulation Run Time

4-1-5. Click **OK**.

4-1-6. Select **Simulation** > **Run Simulation** in the Flow Navigator.

4-1-7. Select **Run Behavioral Simulation**.

© Copyright 2019 Xilinx

4-2. (Optional): Remove the unused signals so that only the following signals are present:

- **clk_pin**
- **rst_pin**
- **sel_pin**
- **led_pins[7:0]**
- **data_to_send[7:0]**
- **data_sent**
- **rxd_clk_rx**

4-2-1. Select the signals to remove and press the **<Delete>** key.

4-3. In the simulation waveform, zoom to full screen and zoom in on the first value of led_pins.

4-3-1. Click the **Zoom Fit** ( ) icon to zoom to full screen.

4-3-2. Highlight the **led_pins[7:0]** signal by clicking it once. The waveform will be emphasized.

4-3-3. Right-click **led_pins[7:0]** and **data_to send[7:0]** value column and select **Radix** > **ASCII**.

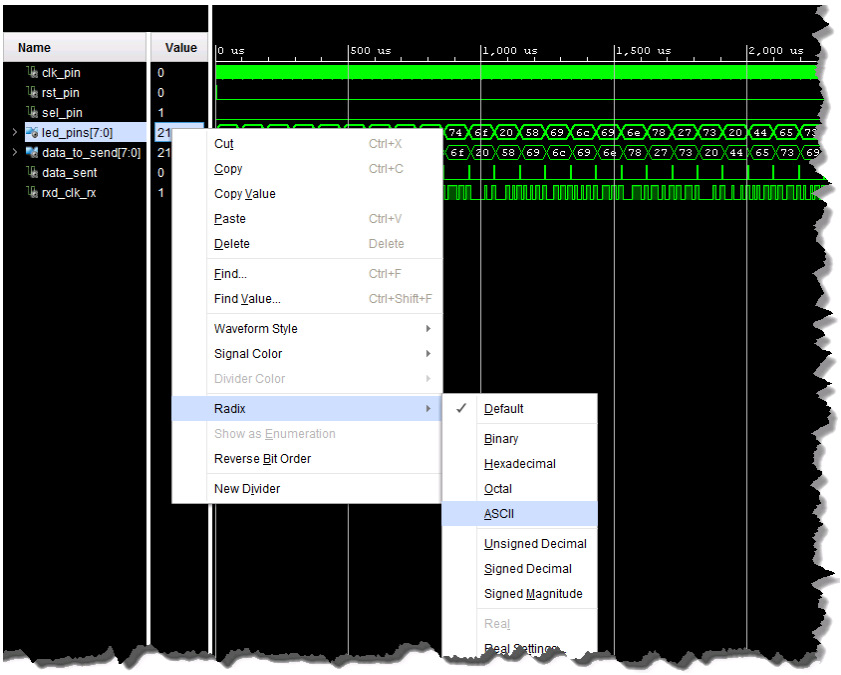


Figure 6-4: Changing the Radix of the Signal

4-3-4. Right-click **led_pins[7:0]** and select **Signal Color**.

4-3-5. Select **Blue** from the provided palette.

© Copyright 2019 Xilinx

Question 1

What are the advantages of changing a signal's radix and color?

4-3-6. Click the **Go To Time 0** ( ) icon in the horizontal toolbar of the waveform viewer to move the cursor to time zero.

4-3-7. Highlight the **led_pins[7:0]** signal again by clicking it.

4-3-8. Click the **Next Transition** ( ) icon to advance to the next transition of **led_pins[7:0]**.

The simulator considers time 0 as the first transition.

4-3-9. Click the **Next Transition** ( ) icon once more to get to the transition near 95.385 us.

4-3-10. Click the **Zoom In** ( ) icon to zoom in. Zoom to the point where the values are clearly seen and that both the preceding value of led_pins is also visible.

You may need to pan to get this point visible.

4-4. Zoom out to read the message.

4-4-1. Click the **Zoom Out** ( ) icon several times so that the message is clearly seen.

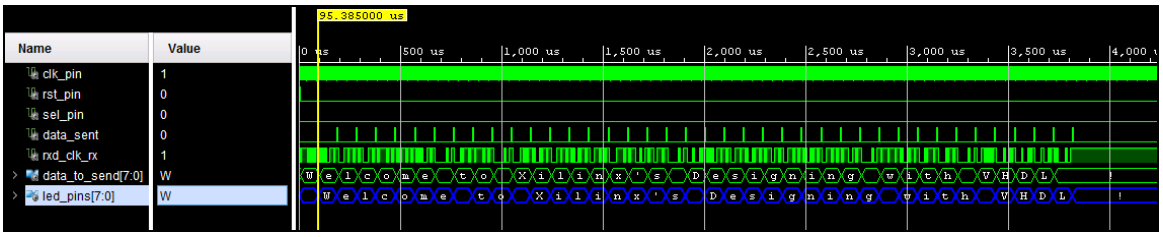


Figure 6-5: Complete Message in the Waveform

Question 2

Does the received message match the sent message? How can you tell?

© Copyright 2019 Xilinx

4-5. Simulate for an additional one millisecond and observe the sel_pin changing to a one and the output of led_pins at this point.

4-5-1. Enter **1 ms** in the duration field.

Figure 6-6: Vivado Simulator Control Toolbar

4-5-2. Click the icon in the horizontal tool bar to run the simulation for 1 ms.

4-5-3. Select **sel_pin** from the Signal Name list and click the **Next Transition** icon ( ).

You may need to pan or zoom out to see the transition.

This should take you to the point where sel_pin transitions from a zero to a one.

4-5-4. Pan and zoom as necessary to view the output of the LED_pins after sel_pin transitions high.

Question 3

What is the value of LED_pins when sel_pin is asserted?

Hint: You may have to change the radix of led_pins[7:0] to binary.

_____

_____

_____

4-6. Close the simulation.

4-7. Close the Vivado Design Suite.

© Copyright 2019 Xilinx

[Optional]: Clean up the file system.

Some systems 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. Both processes will recursively delete all of the files in the `$basicTestbench` directory.

4-8. Clean up the VirtualBox file system via the GUI.

4-8-1. Using the graphical browser (Windows: press the <**Windows**> key + <**E**>; Linux: press <**Ctrl** + **N**>), navigate to `$basicTestbench`.

4-8-2. Select **`basicTestbench`**.

4-8-3. Press <**Delete**>.

4-9. Clean up the VirtualBox file system via the command line.

4-9-1. Open a terminal window (Windows: press the <**Windows**> key + <**R**>, then enter **cmd**; Linux: press <**Ctrl** + **Alt** + **T**>)

4-9-2. Enter the following command to delete the contents of the workspace:

[Windows users]: **`rd /s /q $basicTestbench`**

[Linux users]: **`rm -rf $basicTestbench`**

Summary

This lab walked you through the creation of stimulus for the reset, clock, and sel_pin signals using both concurrent statements and processes.

You also learned more of the basic navigational skills and display features of the Vivado simulator.

© Copyright 2019 Xilinx

Answers

1. What are the advantages of changing a signal's radix and color?

   Changing a signal's radix enables you to view the waveform in its native, or most meaningful, representation. Because binary is one of the longest reorientations, virtually all other displays yield shorter labels, enabling you to see more of the waveform at one time.

   Changing signal colors allows you to highlight or apply special meaning to signals. For example, signals directly connected to a particular module might be shaded red and another module, blue.

2. Does the received message match the sent message? How can you tell?

   There are two ways of seeing that the message was sent.

   Each data_to_send is shown during the serial transmission and is complete when the data_sent signal asserts for one clock period. Each transmitted character can be manually compared to each received character, which is presented slightly later on led_pins.

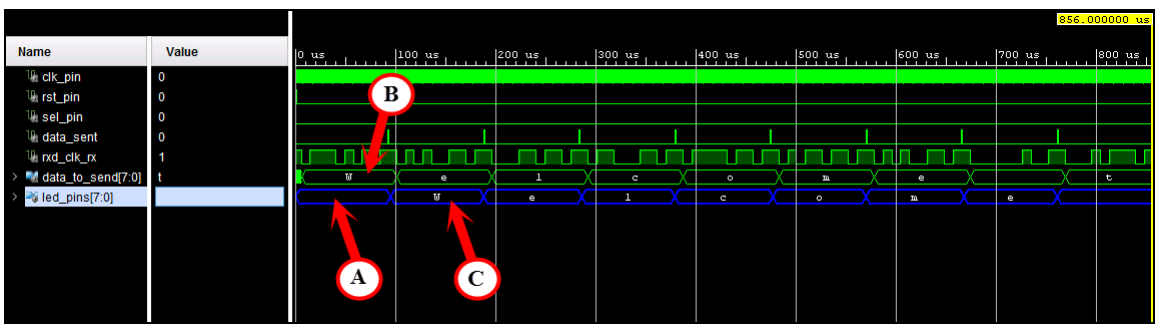


Figure 6-7: Manually Verifying the Character Sequence

'A' – Prior to the transmission of the first character, the output of the UUT is undefined and is shown (in ASCII mode) as a null character.

'B' – The data generator produces the first character and begins transmitting it to the UUT.

'C' – After several clock cycles, the received character is presented on led_pins. Because this character matches the one sent (indicated by 'B'), you can consider this a successful transmission of this character.

© Copyright 2019 Xilinx

The second message requires you to open the uart_led_simple_tb.vhd file and look at the message passed into the tb_uart_driver:

```
tb_uart_driver_i0: tb_uart_driver
    generic map (BIT_PERIOD => BIT_PERIOD, MESSAGE => "Welcome to Xilinx's Designing with VHDL class!")
    port map (  rst_clk_rx        => rst_pin,
                data_to_send      => data_to_send,
                data_sent         => data_sent,
                data_out_serial   => rxd_clk_rx
               );
```

From inspection, the message that appears in the waveform matches that specified in the generic.

3. What is the value of LED_pins when sel_pin is asserted?

   The value changes from '!' to 0 several clocks after sel_pin is asserted. Note that the radix was switched back to binary to avoid any ambiguity in the display of the ASCII character. Non-printing ASCII characters are represented as ' '.

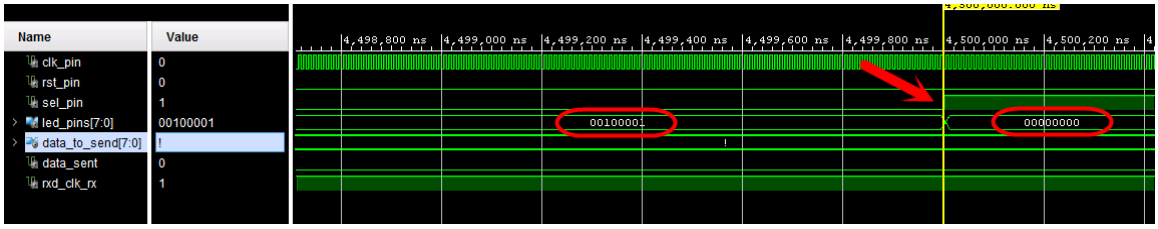


Figure 6-8: Waveform at Assertion of sel_pin

© Copyright 2019 Xilinx

Coding the Stimulus

Procedure or concurrent statement that generates a 100-MHz clock:

Concurrent (RTL style):

```
clk_pin <= not clk_pin after CLK_RX_PERIOD / 2;
```

OR

Process (behavioral style):

```
makeClk_rx: process
   begin
      clk_pin <= '0';
      wait for CLK_RX_PERIOD / 2;
      clk_pin <= '1';
      wait for CLK_RX_PERIOD * 0.5;
end process makeClk_rx;
```

Process or concurrent statement that generates an active high reset for 100 clock pulses:

Concurrent (RTL style):

```
rst_pin <= '1', '0' after CLK_RX_PERIOD * 100;
```

OR

Process (behavioral style):

```
make_reset: process
   begin
      rst_pin <= '1';                                   -- assert reset
      wait for CLK_RX_PERIOD * 100;
      rst_pin <= '0';
      wait;
   end process make_reset;
```

Process or concurrent statement that holds select low for 4.5 ms, then switches to Channel 2 (select high):

Concurrent (RTL style):

```
sel_pin <= '0', '1' after 4.5 ms;
```

-OR

Process (behavioral style):

```
make_sel: process
```


© Copyright 2019 Xilinx

```
begin
        sel_pin <= '0';
        wait for 4.5 ms;
        sel_pin <= '1';
        wait;
  end process make_sel;
```

Verifying the connections to the tb_uart_driver:

```
        -- Instantiate the data generator - uses the predefined character sequence and
transmits bit-by-bit
        tb_uart_driver_i0: tb_uart_driver
            generic map (BIT_PERIOD => BIT_PERIOD, MESSAGE => "Welcome to Xilinx's
Designing with VHDL!")
            port map (   rst_clk_rx          => rst_pin,
                         data_to_send        => data_to_send,
                         data_sent           => data_sent,
                         data_out_serial     => rxd_clk_rx
                       );
```

Connecting the UUT to signal sources and sinks:

```
UUT: uart_led
generic map (CLOCK_RATE => CLOCK_RATE,          -- generics defined in entity
             BAUD_RATE    =>    BAUD_RATE
            )
  port map ( clk_pin          => clk_pin,
             rst_pin          => rst_pin,
             rxd_pin          => rxd_clk_rx,
             sel_pin          => sel_pin,
             led_pins         => led_pins
           );
```


© Copyright 2019 Xilinx

Lab 7: Using Functions

2019.1

Abstract

This lab demonstrates how to create functions in a design while coding with VHDL.

This lab should take approximately 45 minutes.

Objectives

After completing this lab, you will be able to:

- Create a new function
- Modify and use the created function

Introduction

Functions are helpful when there is a need for code repetition.

The string_utilities_sim_pkg package provides a number of string functions that are not synthesizable. This package provides additional string support through a series of constants, functions, and packages.

Functions that are supported are:

- function strAutoResize(s: String) return String – Takes the passed string and places the printing characters into a new string that is the proper length. For example, if a string were defined with a size of 32 characters and string(1) is assigned an 'A', if this were printed (using the write/writeline capabilities of TEXTIO) it would appear as 31 spaces followed by an 'A'. If strAutoResize is called first, it will return a string of length 1—just the right size for 'A'.
- function substr(str: String; startingPoint, endingPoint : integer) return String – Returns a string of the appropriate size containing the portion of the string from starting point to ending point.
- function strlen(s : String) return integer – Returns the length of the printable characters is the passed string, not the size of the string container.
- function strdel(str: String; startingPoint, endingPoint : integer) return String – Removes characters between startingPoint and endingPoint in an appropriately sized container.
- function strins(str1, str2 : String; startingPoint : integer) return String – The converse of strdel—it will insert str2 into str1 beginning at startingPoint.
- function strcat(s1, s2: String) return String – Similar to the concatenation operator '&,' but it first properly resizes both S1 and S2 before joining the two strings.

© Copyright 2019 Xilinx

- function strpos(s : String; c : character) return integer – Returns the first occurrence of the character c within the string.

In this lab, you will be creating and adding two new functions. The first will return the position of the next occurrence of c within s, and the other will always return the last position of the last occurrence of c in s.

To practice writing synthesizable functions, you will also add two additional functions to the package: one that will 'and' all the bits of a std_logic_vector together, and another that will 'or' them together.

Understanding the Lab Environment

The labs and demos provided in this course are designed to run under the provided Linux platform. The provided Linux platform has a customized lab environment. These customizations have been made to simplify and enhance your learning experience. Many labs and demos can be successfully executed in the Windows environment as well. Topics including PetaLinux, Yocto, and QEMU are generally not supported under Windows.

The instructions, for the sake of brevity, are expressed using the Linux notation. This includes the forward slash ('/') as the hierarchy separator instead of the Windows backslash ('\'). For students who want to run under Windows, it is their responsibility to use the correct hierarchy separator.

Windows users can run Linux using the VirtualBox tool from Oracle. Details on how to acquire and install this tool can be found in the lab setup guide. The Xilinx Customer Education group offers a preconfigured virtual machine image and includes all the tools necessary to run the labs. Some tools and IP do require licenses, which are the responsibility of the student.

Several environment variables, described in detail in the lab setup guide, which also discusses how you can customize the environment for your situation, are provided for ease of use. Most notably, many environment variables have been defined to reduce the amount of typing you need to do when entering paths.

These environment variables are:

- XILINX_PATH - points to the installation directory for the Xilinx tools. Tools such as the Vivado Design Suite and SDK can be found here.
- PETALINUX_PATH - points to the installation directory for the Xilinx PetaLinux tools. Often this is within XILINX_PATH, but some choose to install it elsewhere.
- TRAINING_PATH - points to the space allocated for students to work through their labs. This directory includes prebuilt images and starting points for the labs and demos.
- functions - points to the specific name of the topic within the TRAINING_PATH directory.

  This lab uses $functions to indicate the directory for this lab, which is equivalent to $TRAINING_PATH/functions.

These environment variables work well in the Linux terminal and the Vivado Design Suite and scripts, but not in Eclipse-based tools such as the SDK tool and the SDx™ environment. When

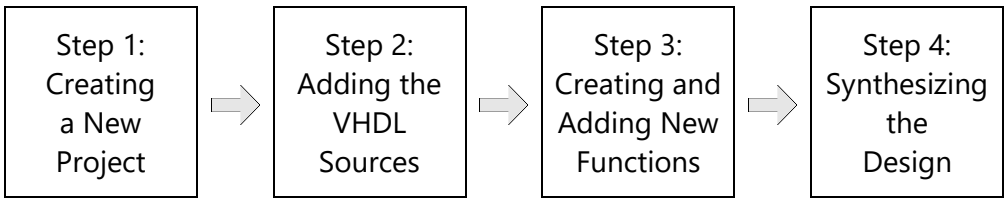
© Copyright 2019 Xilinx

using the Vivado tools, you can use $functions and the tools will automatically expand the environment variable for you.

However, when you see the notation $functions/workspace in SDK or the SDx environment, you should interpret it in its expanded form. The expanded form can be found by opening a terminal window and enter "echo $functions".

This lab assumes that you are starting with a Linux environment. If you are working on a Windows system, refer to the lab setup guide to see how to use the Oracle VirtualBox application to run the Xilinx Customer Education pre-configured Ubuntu Linux virtual machine.

General Flow



Creating a New Project Step 1

1-1. Launch the Vivado Design Suite.

If you do not recall how to perform this task, refer to the "Launching the Vivado Design Suite" section under Vivado Design Suite Operations in the *Lab Reference Guide*.

1-2. Create a new Vivado Design Suite project named *functions* and locate it in the following directory.

Browse to the `$functions/lab/KCU105` directory.

Target the project for the Kintex UltraScale KCU105 Evaluation Platform evaluation board.

If you do not recall how to perform this task, refer to the "Creating a Blank Vivado Design Suite Project" section in the *Lab Reference Guide*.

From the settings in the Flow Navigator, change the target language to VHDL.

© Copyright 2019 Xilinx

Adding the VHDL Sources — Step 2

In this step, you will add the VHDL sources to the project.

2-1. Add the VHDL sources.

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

The Add Sources Wizard opens.

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

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

2-1-4. Click the **Plus** (+) icon and select **Add Files**.

2-1-5. Press the <**Ctrl**> and select all the files listed below:

- uart_led.vhd
- reset_bridge.vhd
- meta_harden.vhd
- uart_rx.vhd
- uart_rx_ctl.vhd
- uart_baud_gen.vhd
- LED_manager.vhd
- register8.vhd
- UART_LED_pkg.vhd

2-1-6. Click **OK**.

2-1-7. Click **Finish** to confirm adding the selected files to the project with the default associations and libraries.

2-1-8. Make certain that all the files are added to the xil_defaultlib library and not utilities_lib.

2-2. Add the simulation sources.

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

The Add Sources Wizard opens.

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

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

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

2-2-5. Select **string_utilities_sim_pkg.vhd**.

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

© Copyright 2019 Xilinx

2-2-7. Click the **xil_defaultlib** entry in the Library field in the Add or Create Simulation Sources dialog box (highlighted in the figure below) and enter **utilities_lib**.

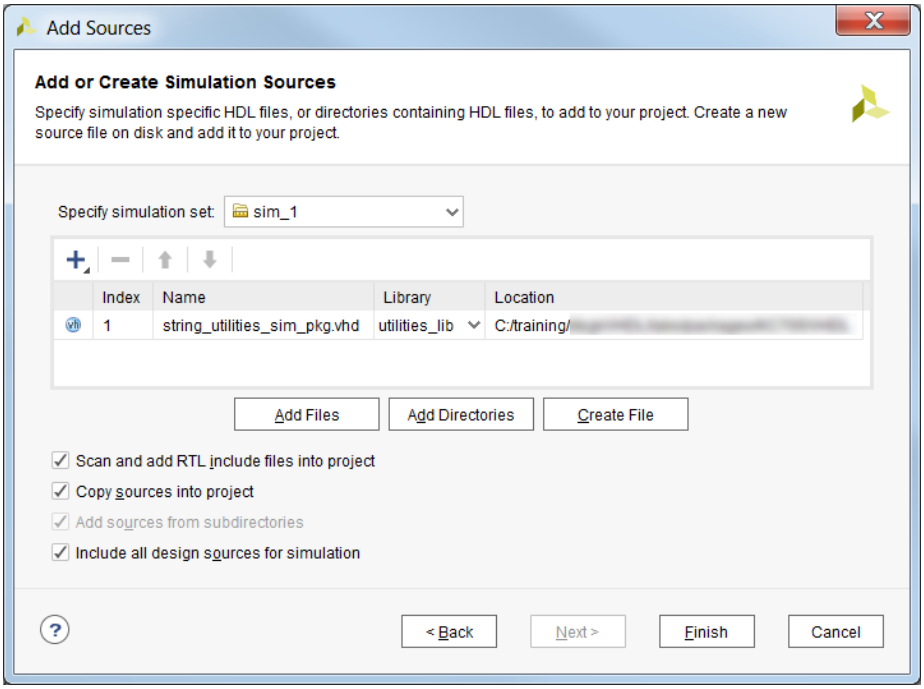


Figure 7-1: Associating the Library

The string_utilities_sim_pkg can only be used in simulation.

2-2-8. Click **Finish** to close the Add and Create Simulation sources dialog box.

© Copyright 2019 Xilinx

Creating and Adding Functions to Existing Packages Step 3

Whenever there is a need for performing the same activity again and again, that functionality can be added in functions and these functions can be declared in the code.

Here you will be adding functions to the existing packages.

3-1. Open the existing package UART_LED_pkg for editing.

3-1-1. Select the **Sources** view.

3-1-2. Select the **Libraries** tab see all the files added to the project.

3-1-3. Expand **Design Sources** > **VHDL** > **xil_defaultlib**.

3-1-4. Double-click **UART_LED_pkg.vhd**.

UART_LED_pkg.vhd opens in the editor.

Question 1

What does this template contain?

_____

_____

3-2. Add the function declarations for and_bits_together and or_bits_together (takes a std_logic_vector and returns a std_logic).

Although a suggested code solution can be found in the Answers section, you should make every effort to code this yourself.

These functions take one input argument of type std_logic_vector and returns a single value of type std_logic_vector.

3-2-1. Declare the functions for *and_bits_together* and *or_bits_together*.

3-2-2. Code the outline for both and_bits_together and or_bits_together in UART_LED_pkg.vhd, using the provided templates under the package body.

Each function must return a value.

3-2-3. Define a variable of the proper type and, using a return statement, return this value.

3-2-4. Initialize this return value.

Hint: In order for and_bits_together to evaluate true, the initial value of the value being returned must be a '1'. This is not the case for or_bits_together.

© Copyright 2019 Xilinx

3-2-5. Create an appropriate loop to perform the AND-ing and OR-ing.

Hint: Use the functions that you defined in the simulation package as a guide.

3-2-6. Select **File** > **Text Editor** > **Save File**.

Question 2

Based on the descriptions above, what must be entered into the package portion of this source file?

3-3. Open string_utilties_sim_pkg.vhd for editing and add two functions:

- **strposnext(s - string, c - character, pos - integer) where s is the string in which to locate the next occurrence of character c beginning from position pos. This function should return a non-zero integer if successful, 0 if unsuccessful.**
- **strposlast(s – string, c – character) where the position of the last occurrence of character c within s is returned. 0 is returned if c does not exist within s.**

Although suggested code solutions can be found in the Answers section, you should make every effort to code these yourself.

3-3-1. Select the **Libraries** tab in the Sources view.

3-3-2. Expand **Simulation-Only Sources**.

3-3-3. Double-click **string_utilities_sim_pkg.vhd** under **sim_1** > **VHDL** > **utilities_lib** > **Unreferenced**.

3-3-4. Write the outline for both of these functions, using either the existing functions as a template or selecting **Tools** > **Language Templates** > **VHDL** > **Common Constructs** > **User Defined Functions & Procedures** > **Function Body**.

3-3-5. Examine the code for strpos.

Notice the use of the predefined attribute ‘length to automatically adjust to the proper length of string s.

3-3-6. Code the function body for strposlast.

Hint: The last occurrence is the same as the first occurrence if the loop range is reversed.

© Copyright 2019 Xilinx

3-3-7. Code the function body for strposnext.

Hints:

- This is the same as strpos except that the starting point is not 1.
- Both of these above functions take one string and one character as an input.
- Returns an integer value.
- A for loop can be added to cover the entire range of a string.

3-3-8. Select **File** > **Text Editor** > **Save File**.

Synthesizing the Design — Step 4

Here you will observe the newly created functions and synthesize the design.

4-1. Observe the UART_led.vhd file.

4-1-1. Observe how the UART_led_pkg package has been added in the UART_led file.

```
library xil_defaultlib;
use xil_defaultlib.UART_led_pkg.all;
```

4-2. Synthesize (compile) the design to check for any errors.

4-2-1. Click **Run Synthesis** under Synthesis in the Flow Navigator.

4-2-2. Correct errors as necessary, save the file, and resynthesize.

4-2-3. Verify that the design has been successfully synthesized.

4-2-4. Click **Cancel** if the Synthesis Completed dialog box appears.

4-3. Close the Vivado Design Suite.

4-3-1. Select **File** > **Exit**.

The Exit Vivado dialog box opens.

Figure 7-2: Exit Vivado Dialog Box

4-3-2. Click **OK**.

© Copyright 2019 Xilinx

[Optional]: Clean up the file system.

Some systems 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. Both processes will recursively delete all of the files in the `$functions` directory.

4-4. Clean up the VirtualBox file system via the GUI.

4-4-1. Using the graphical browser (Windows: press the <**Windows**> key + <**E**>; Linux: press <**Ctrl** + **N**>), navigate to `$functions`.

4-4-2. Select **`functions`**.

4-4-3. Press <**Delete**>.

4-5. Clean up the VirtualBox file system via the command line.

4-5-1. Open a terminal window (Windows: press the <**Windows**> key + <**R**>, then enter **cmd**; Linux: press <**Ctrl** + **Alt** + **T**>)

4-5-2. Enter the following command to delete the contents of the workspace:

[Windows users]: **`rd /s /q $functions`**

[Linux users]: **`rm -rf $functions`**

Summary

This lab walked you through the process of creating functions, both declaring and defining the functions. These functions were then added to packages.

Finally, the design was synthesized.

© Copyright 2019 Xilinx

Answers

1. What does this template contain?

   The template contains both the package template and the package body template. Patterns are provided for types, constants, functions, and procedures declarations in the package. Functions and procedures templates are provided in the package body.

   Notice that *none* of the information is filled in — not even the package name.

2. Based on the descriptions above, what must be entered into the package portion of this source file?

   The function declaration must be entered as follows in the package declaration area:

   function strposnext(s : *String*; c : *character*; pos : *integer*) *return integer*;

   function strposlast(s: *String*; c : *character*) *return integer*;

   These can be entered anywhere within the package declaration area.

© Copyright 2019 Xilinx

Creating a New Package

Function declaration for *and_bits_together* and *or_bits_together* functions:

```
   -- Declare functions and procedure
  function and_bits_together (signal slv : in std_logic_vector) return std_logic;
  function or_bits_together  (signal slv : in std_logic_vector) return std_logic;
```

Constant declaration for gnd_i constant:

```
     -- Declare constants
constant gnd_i : std_logic_vector(7 downto 0) := (others=>'0');
```

Function body for *and_bits_together* and *or_bits_together* functions:

```
function and_bits_together (signal slv : in std_logic_vector) return std_logic is
    variable return_sl : std_logic := 'U';
begin
    return_sl := '1';
    andAll: for i in 0 to slv'length-1 loop
                return_sl := return_sl AND slv(i);
            end loop;
    return return_sl;
end function and_bits_together;

function or_bits_together (signal slv : in std_logic_vector) return std_logic is
   variable return_sl : std_logic := 'U';
begin
    return_sl := '0';
    andAll: for i in 0 to slv'length-1 loop
                return_sl := return_sl OR slv(i);
            end loop;
    return return_sl;
end function or_bits_together;
```


© Copyright 2019 Xilinx

Modifying the Existing Packages

Outlines for strposnext and strposlast:

```
-- ************************************************************
--  Function: strposnext
--  Inputs  : String, character, integer
--  Outputs : integer
--  Description : finds the first occurances after the pos position
--    of character in string and returns its integer position.
--    0 if character not found
-- ************************************************************
function strposnext (s   : string;
                     c   : character;
                     pos : integer) return integer is
  -- subprogram_declarative_items (constant declarations, variable declarations, etc.)
   begin
     -- function body
   end strposnext;

-- ************************************************************
--  Function: strposlast
--  Inputs  : String, character
--  Outputs : integer
--  Description : finds the last occurances
--    of character in string and returns its integer position.
--    0 if character not found
-- ************************************************************
function strposlast (s   : string;
                     c   : character) return integer is
  -- subprogram_declarative_items (constant declarations, variable declarations, etc.)
  begin
     -- function body
  end strposlast;
```

Function body for strposlast:

```
-- ************************************************************
--  Function: strposlast
--  Inputs  : String, character
--  Outputs : integer
--  Description : finds the last occurances
--    of character in string and returns its integer position.
--    0 if character not found
-- ************************************************************
function strposlast (s   : string;
                     c   : character) return integer is
  begin
      spos: for i in s'length downto 1 loop
         if (s(i) = c) then
            return i;
         end if;
      end loop;
      return 0;
  end strposlast;
```

Function body for strposnext.

```
-- ************************************************************
--  Function: strposnext
--  Inputs  : String, character, integer
--  Outputs : integer
--  Description : finds the first occurances after the pos position
--    of character in string and returns its integer position.
--    0 if character not found
-- ************************************************************
function strposnext (s   : string;
                     c   : character;
                     pos : integer) return integer is
   begin
      spos: for i in pos to s'length loop
         if (s(i) = c) then
            return i;
         end if;
      end loop;
      return 0;
   end strposnext;
```


© Copyright 2019 Xilinx

Lab 8: Building Your Own Package

2019.1

Abstract

This lab demonstrates how to define, declare, and use the packages in a design while coding with VHDL.

This lab should take approximately 45 minutes.

Objectives

After completing this lab, you will be able to:

- Modify and build upon previous packages
- Create and use a custom package

Introduction

Packages provide a relatively pain-free mechanism for reusing small pieces of useful code. As you are probably well aware of by now, although VHDL provides basic string handling capabilities, they are not fully featured.

To this end, two incomplete packages have been provided. These packages provide additional string support through a series of constants, functions, and packages. The string_utilities_sim_pkg package provides a number of string functions that are not synthesizable.

Functions that are currently supported are:

- function strAutoResize(s: String) return String – Takes the passed string and places the printing characters into a new string that is the proper length. For example, if a string were defined with a size of 32 characters and string(1) is assigned an 'A', if printed (using the write/writeline capabilities of TEXTIO) it would appear as 31 spaces followed by an 'A'. If strAutoResize is called first, it will return a String of length 1—just the right size for 'A'.
- function substr(str: String; startingPoint, endingPoint : integer) return String – Returns a string of the appropriate size containing the portion of the string from starting point to ending point.
- function strlen(s : String) return integer – Returns the length of the printable characters is the passed string, not the size of the string container.
- function strdel(str: String; startingPoint, endingPoint : integer) return String – Removes characters between startingPoint and endingPoint in an appropriately sized container.

© Copyright 2019 Xilinx

- function strins(str1, str2 : String; startingPoint : integer) return String – The converse of strdel—it will insert str2 into str1 beginning at startingPoint.
- function strcat(s1, s2: String) return String – Similar to the concatenation operator '&,' but it first properly resizes both S1 and S2 before joining the two strings.
- function strpos(s : String; c : character) return integer – Strpos returns the first occurrence of the character c within the string.

You will modify existing package by adding functions to this package.

You will also create a custom package for the UART_LED design that moves all of the component declarations into the package. This is often done to both reduce the clutter of the source code as well as provide a single point of declaration.

A single point of declaration serves as a time-saving device during code development. Should any aspect of a module's entity change, it only needs to be updated in the package and in any instantiation of that module.

Understanding the Lab Environment

The labs and demos provided in this course are designed to run under the provided Linux platform. The provided Linux platform has a customized lab environment. These customizations have been made to simplify and enhance your learning experience. Many labs and demos can be successfully executed in the Windows environment as well. Topics including PetaLinux, Yocto, and QEMU are generally not supported under Windows.

The instructions, for the sake of brevity, are expressed using the Linux notation. This includes the forward slash ('/') as the hierarchy separator instead of the Windows backslash ('\'). For students who want to run under Windows, it is their responsibility to use the correct hierarchy separator.

Windows users can run Linux using the VirtualBox tool from Oracle. Details on how to acquire and install this tool can be found in the lab setup guide. The Xilinx Customer Education group offers a preconfigured virtual machine image and includes all the tools necessary to run the labs. Some tools and IP do require licenses, which are the responsibility of the student.

Several environment variables, described in detail in the lab setup guide, which also discusses how you can customize the environment for your situation, are provided for ease of use. Most notably, many environment variables have been defined to reduce the amount of typing you need to do when entering paths.

These environment variables are:

- XILINX_PATH - points to the installation directory for the Xilinx tools. Tools such as the Vivado Design Suite and SDK can be found here.
- PETALINUX_PATH - points to the installation directory for the Xilinx PetaLinux tools. Often this is within XILINX_PATH, but some choose to install it elsewhere.
- TRAINING_PATH - points to the space allocated for students to work through their labs. This directory includes prebuilt images and starting points for the labs and demos.

© Copyright 2019 Xilinx

- packages - points to the specific name of the topic within the TRAINING_PATH directory.

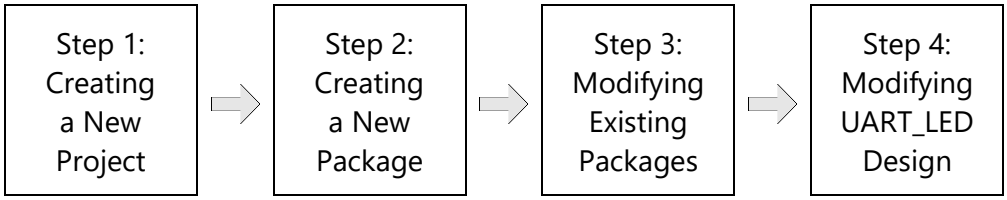
  This lab uses $packages to indicate the directory for this lab, which is equivalent to $TRAINING_PATH/packages.

These environment variables work well in the Linux terminal and the Vivado Design Suite and scripts, but not in Eclipse-based tools such as the SDK tool and the SDx™ environment. When using the Vivado tools, you can use $packages and the tools will automatically expand the environment variable for you.

However, when you see the notation $packages/workspace in SDK or the SDx environment, you should interpret it in its expanded form. The expanded form can be found by opening a terminal window and enter "echo $packages".

This lab assumes that you are starting with a Linux environment. If you are working on a Windows system, refer to the lab setup guide to see how to use the Oracle VirtualBox application to run the Xilinx Customer Education pre-configured Ubuntu Linux virtual machine.

General Flow



Creating a New Project — Step 1

1-1. Launch the Vivado Design Suite.

If you do not recall how to perform this task, refer to the "Launching the Vivado Design Suite" section under Vivado Design Suite Operations in the *Lab Reference Guide*.

1-2. Create a new Vivado Design Suite project named *packages* and locate it in the following directory.

Browse to the `$packages/lab/KCU105` directory.

Target the project for the Kintex UltraScale KCU105 Evaluation Platform evaluation board.

If you do not recall how to perform this task, refer to the "Creating a Blank Vivado Design Suite Project" section in the *Lab Reference Guide*.

From the settings in the Flow Navigator, change the target language to VHDL.

© Copyright 2019 Xilinx

Creating a New Package Step 2

Packages are often created with the intention of reuse. Typically, these packages are located in libraries that are shared among multiple designers. However, it is often practical to have packages for a specific project. These packages typically include constants, types, functions/procedures, and component declarations that are used throughout the design. Placing this type of information in a package reduces the risk of creating a redefined constant, type, or other item.

This step will walk you through the creation of a package for the UART_LED design. The package will contain the component declarations, constants, and functions for the UART_LED design.

2-1. Because you will be creating a package for UART_LED, load the files for UART_LED into the project. They are located in your working directory:

- **uart_led.vhd**
- **reset_bridge.vhd**
- **meta_harden.vhd**
- **uart_rx.vhd**
- **uart_rx_ctl.vhd**
- **uart_baud_gen.vhd**
- **LED_manager.vhd**
- **register8.vhd**
- **UART_LED_pkg.vhd**

Make sure that these files are all loaded into the xil_defaultlib library, not utilities_lib.

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

The Add Sources Wizard opens.

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

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

2-1-4. Click the **Plus** (+) icon and select **Add Files**.

© Copyright 2019 Xilinx

2-1-5. Press the <**Ctrl**> key and select all the files listed above.

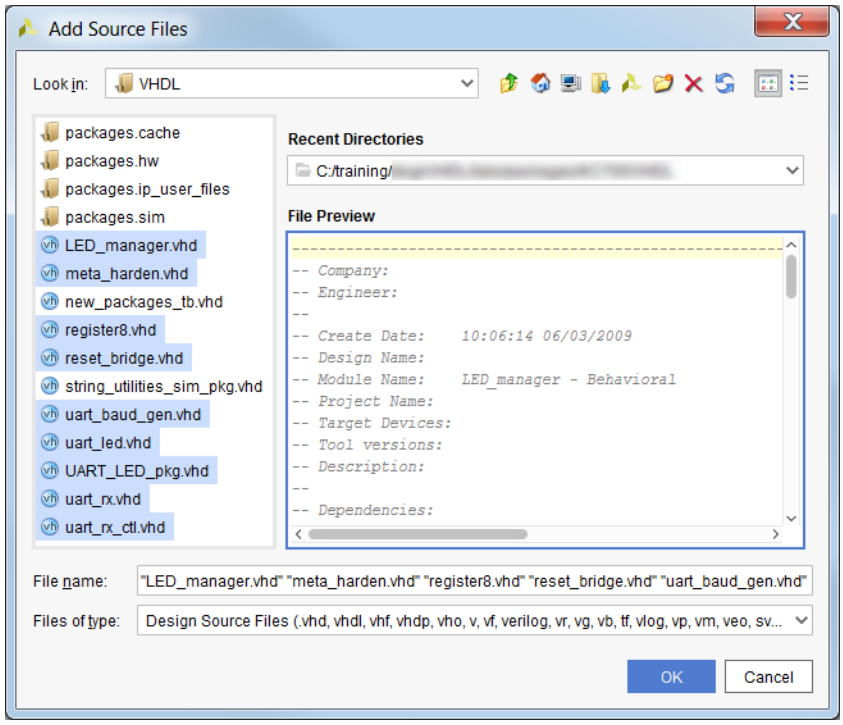


Figure 8-1: Loading the Files for the UART_led Design

2-1-6. Click **OK**.

2-1-7. Click **Finish** to confirm adding the selected files to the project with the default associations and libraries.

2-1-8. Make certain that all the files are added to the xil_defaultlib library and not utilities_lib.

2-2. Open the package named UART_LED_pkg.vhd for editing.

2-2-1. Select the **Sources** view.

2-2-2. Select the **Libraries** tab see all the files added to the project.

2-2-3. Expand **Design Sources** > **VHDL** > **xil_defaultlib** > **Unreferenced(1)**.

2-2-4. Double-click **UART_LED_pkg.vhd**.

UART_LED_pkg.vhd opens in the editor.

Question 1

What does this template contain?

_____

_____

© Copyright 2019 Xilinx

2-3. Add the following information to the package:

- **All of the component declarations defined in uart_led.vhd.**
- **Function declarations for and_bits_together and or_bits_together (takes a std_logic_vector and returns a std_logic).**
- **Constant declarations for an 8-bit ground signal (gnd_i).**

Although a suggested code solution can be found in the Answers section, you should make every effort to code this yourself.

2-3-1. Select the **Sources** view.

2-3-2. Select the **Hierarchy** tab to switch to the Hierarchy view.

2-3-3. Double-click **uart_led.vhd** to open it in the text editor.

2-3-4. Select and highlight all the component definitions.

2-3-5. Right-click and select **Cut**.

You can also press <**Ctrl** + **X**>.

2-3-6. Select the **UART_LED_pkg.vhd** tab to switch to the package source.

2-3-7. Position the cursor after the "package UART_LED_pkg is" line.

2-3-8. Right-click and select **Paste**.

You can also press <**Ctrl** + **V**>.

2-3-9. Cut and paste the gnd_i constant from uart_led. This creates a constant named *gnd_i* that is an 8-bit std_logic_vector initialized to '0'. You can also use the constant template provided in the package.

2-3-10. Declare functions for *and_bits_together* and *or_bits_together*.

2-3-11. Uncomment and observe the previously added function definitions.

2-3-12. Comment out or remove any remaining template that is not being used.

2-3-13. Select **File** > **Text Editor** > **Save File**.

© Copyright 2019 Xilinx

Modifying Existing Packages Step 3

You will load two existing packages into a library and mark them for compilation. You will then add one function to each and verify that the functions are behaving as expected.

3-1. Load string_utilities_sim_pkg into a new library named utilities_lib. Make certain that both libraries will be compiled when changes are made to them.

3-1-1. Click **Add Sources** under Project Manager in the Flow Navigator.

The Add Sources Wizard opens.

3-1-2. Select **Add or create simulation sources**.

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

3-1-4. Click the **Plus** (+) icon and select **Add Files**.

3-1-5. Select **string_utilities_sim_pkg.vhd**.

3-1-6. Click **OK**.

3-1-7. Click the **xil_defaultlib** entry in the Library field in the Add or Create Simulation Sources dialog box (highlighted in the figure below) and enter **utilities_lib**.

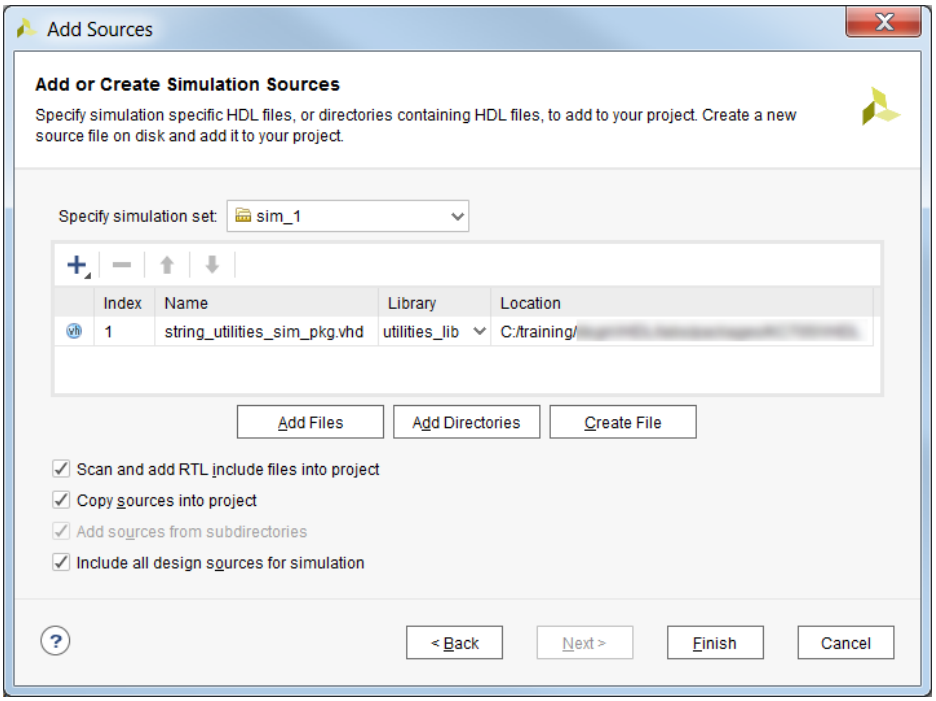


Figure 8-2: Associating the Library

The string_utilities_sim_pkg can only be used in simulation.

3-1-8. Click **Finish** to close the Add and Create Simulation sources dialog box.

© Copyright 2019 Xilinx

3-2. Open string_utilties_sim_pkg.vhd for editing and add two functions:

- **strposnext(s - string, c - character, pos - integer) where s is the string in which to locate the next occurrence of character c beginning from position pos. This function should return a non-zero integer if successful, 0 if unsuccessful.**
- **strposlast(s – string, c – character) where the position of the last occurrence of character c within s is returned. 0 is returned if c does not exist within s.**

Although suggested code solutions can be found in the Answers section, you should make every effort to code these yourself.

3-2-1. Select the **Libraries** tab in the Sources view.

3-2-2. Expand **Simulation-Only Sources**.

3-2-3. Double-click **string_utilities_sim_pkg.vhd** under **sim_1** > **VHDL** > **utilities_lib** > **Unreferenced**.

Question 2

Based on the descriptions above, what must be entered into the package portion of this source file?

_____

_____

_____

3-2-4. Add your answer to the previous question about what the template contains to the code to define the function in the package.

Question 3

After adding your answer to the previous question about what the template contains to the code, do you think this is sufficient to implement these functions?

_____

_____

_____

3-2-5. Write the outline for both of these functions, using either the existing functions as a template or selecting **Tools** > **Language Templates** > **VHDL** > **Common Constructs** > **User Defined Functions & Procedures** > **Function Body**.

© Copyright 2019 Xilinx

3-2-6. Examine the code for strpos.

Notice the use of the predefined attribute ‘length to automatically adjust to the proper length of string s.

3-2-7. Uncomment and observe the previously added function body for strposlast.

Hint: The last occurrence is the same as the first occurrence if the loop range is reversed.

3-2-8. Uncomment and observe the previously added function body for strposnext.

Hint: This is the same as strpos except that the starting point is not 1.

3-2-9. Select **File** > **Text Editor** > **Save File**.

Modifying the UART_LED Design — Step 4

Now that the package is built, you will want to take advantage of its benefits. This step will demonstrate how to modify the non-package portions of your source code to use the new package.

During the creation of the package, you cut-and-pasted the component declarations and constants which is most of the modification that is needed.

Question 4

What other modifications are required to make the package usable by UART_LED?

_____

_____

4-1. Open the UART_led.vhd file.

Although a suggested code solution can be found in the Answers section, you should make every effort to code this yourself.

4-1-1. Add the line of code that opens the library.

This line of code should precede the entity.

4-1-2. Specify the package within the library for use and to indicate that all aspects of the package should be made available.

library xil_defaultlib;

use xil_defaultlib.UART_led_pkg.all;

4-1-3. Select **File** > **Text Editor** > **Save File**.

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בכרטיסיה בית כדי להחיל

4-2. (Optional): If any component declarations or constants remain in UART_LED, delete them.

4-3. Synthesize (Compile) the design to check for any errors.

4-3-1. Click **Run Synthesis** under Synthesis in the Flow Navigator.

4-3-2. Correct errors as necessary, save the file, and resynthesize.

4-3-3. Click **Cancel** if the Synthesis Completed dialog box appears.

4-4. Add the New package (new_packages_tb.vhd) file to the project.

The new_packages_tb.vhd testbench has been provided for you.

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

The Add Sources Wizard opens.

4-4-2. Select **Add or Create Simulation Sources** and click **Next**.

4-4-3. Click the **Plus** (+) icon and select **Add Files**.

4-4-4. Select **new_packages_tb.vhd**.

4-4-5. Click **OK** and then **Finish** to add the file to the project.

4-5. Set the new_packages_tb.vhd as the top-level file for simulation.

4-5-1. Expand **Simulation Sources** in the Hierarchy tab of the Sources view.

4-5-2. Right-click **new_packages_tb.vhd** under **Simulation Sources** > **sim_1** and select **Set as Top**.

4-5-3. Double-click **new_packages_tb** to view it in the text editor.

4-5-4. Explore the testbench.

Question 5

What stimulus does the testbench provide?

_____

_____

_____

Question 6

Is this a robust test of these functions?

_____

_____

© Copyright 2019 Xilinx

Question 7

What is the expected output of this testbench?

_____

_____

_____

4-6. Launch the Vivado simulator to test the new functions.

4-6-1. Select **Simulation** > **Run Simulation** > **Run Behavioral Simulation** in the Flow Navigator.

This will launch the Vivado simulator.

4-6-2. Check the testbench response in the Vivado Tcl console.

Question 8

Do your expectations and the output of this simulation match?

_____

_____

4-7. Close the Vivado Design Suite.

4-7-1. Select **File** > **Exit**.

The Exit Vivado dialog box opens.

Figure 8-3: Exit Vivado Dialog Box

4-7-2. Click **OK**.

© Copyright 2019 Xilinx

[Optional]: Clean up the file system.

Some systems 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. Both processes will recursively delete all of the files in the `$packages` directory.

4-8. Clean up the VirtualBox file system via the GUI.

4-8-1. Using the graphical browser (Windows: press the <**Windows**> key + <**E**>; Linux: press <**Ctrl** + **N**>), navigate to `$packages`.

4-8-2. Select **`packages`**.

4-8-3. Press <**Delete**>.

4-9. Clean up the VirtualBox file system via the command line.

4-9-1. Open a terminal window (Windows: press the <**Windows**> key + <**R**>, then enter **cmd**; Linux: press <**Ctrl** + **Alt** + **T**>)

4-9-2. Enter the following command to delete the contents of the workspace:

[Windows users]: **`rd /s /q $packages`**

[Linux users]: **`rm -rf $packages`**

Summary

This lab walked you through the process of updating a provided package, demonstrating how packages can be placed into libraries, and if the expectation is that they must be modified, how to include them for compilation.

The common task of pulling frequently used component definitions, constants, and types, into a package in the working directory shows how code size can be reduced as well as easing the task of updating changes to modules.0

Finally, this lab demonstrated that testbenches can even be run on non-synthesizable code.

© Copyright 2019 Xilinx

Answers

1. What does this template contain?

   The template contains both the package template and the package body template. Patterns are provided for types, constants, functions, and procedures declarations in the package. Functions and procedures templates are provided in the package body.

   Notice that *none* of the information is filled in — not even the package name.

2. Based on the descriptions above, what must be entered into the package portion of this source file?

   The function declaration must be entered as follows in the package declaration area:

   function strposnext(s : *String*; c : *character*; pos : *integer*) *return integer*;

   function strposlast(s: *String*; c : *character*) *return integer*;

   These can be entered anywhere within the package declaration area, although placing them beneath strpos makes more sense as it keeps similar functions together.

3. After adding your answer to the previous question about what the template contains to the code, do you think this is sufficient to implement these functions?

   No, these entries into the package only declare that these functions exist (or will exist) and what their interface looks like. The behavior of these functions must still be defined.

4. What other modifications are required to make the package usable by UART_LED?

   After the "duplicate" definitions have been removed (which should not be a problem unless there is a difference in the definitions), the only aspect remaining is to indicate to the VHDL file that the package should be used.

   This is easily done via the "use" statement. Because the new package has been created in the work library, a library statement is not required because work is always available. Calling out library work is allowable and has no real effect.

5. What stimulus does the testbench provide?

   The test process "calls" the strpos(), strposlast(), and strposnext{} functions with various stimulus and sends the results to the console.

   The logic process "calls" and_bits_together and or_bits_together with various stimulus and sends the results to the console.

© Copyright 2019 Xilinx

6. Is this a robust test of these functions?

   No, but it covers many of the uses. The code is simple enough that between a light simulation and a visual inspection, the code may be adequate for internal use. More verification cases should be applied before this is released to outside sources.

7. What is the expected output of this testbench?

   The testbench first runs the logic module (the delay after start is 100 ps, rather than the 5 ns of the test process).

   The first activity for this process is to wait 100 ps then assign slv_short to all ones. The 1 fs delay is required for slv_short to take on the assigned value as this is a signal (required by the way the function was written). Recall that signals only take on new values when the process suspends, so this adds a little suspense. The process then calls and_bits_together and or_bits_together, simultaneously passing both functions slv_short. Because these are variables, the values are assigned instantly and are reported via the next statement. And because slv_short is set to '1', the expectation is that the AND-ing of the bits should result in a '1' as a result. The result of the OR-ing should also be a '1'.

   After a short delay, slv_medium is assigned a value and the functions being tested are called again. This time, because only some of the bits are ones, the expectation is that the result of the AND-ing returns a logic 0 and the result of the OR-ing returns a logic '1'.

   Finally, the same process is repeated for slv_long where all the bits are zeroed. In this case, the results of the function calls should both return zeros.

   This process effectively completes prior to the test process producing stimulus.

   First, the test process displays the message which is the 26 characters of the English alphabet. Next, strpos is called with the alphabet and 'C' as argument. Because 'C' is the third letter in the alphabet, the expectation is that strpos will return a 3.

   strposlast is then called, looking for the letter 'X'. Because X is third from the end of the alphabet, the expectation is that this call will return 24. In this case, strposlast and strpos will produce the same result as there is only one character that matches the argument in the message.

   Next, the three functions are called with a new message that is comprised of the letter 'X' every fifth character. strpos should return a 1 because the 'X' is in the first position. strlast should return a 26 because 'X' is the last character in this string. strposnext from the 13th position should return the next highest position of X, which will be in position 16.

8. Do your expectations and the output of this simulation match?

   They should. If they do not, ask for assistance from your instructor.

© Copyright 2019 Xilinx

Creating a New Package

Function declaration for *and_bits_together* and *or_bits_together* functions:

```
 -- Declare functions and procedure
function and_bits_together (signal slv : in std_logic_vector) return std_logic;
function or_bits_together  (signal slv : in std_logic_vector) return std_logic;
```

Constant declaration for gnd_i constant:

```
    -- Declare constants
constant gnd_i : std_logic_vector(7 downto 0) := (others=>'0');
```

Modifying the UART_LED Design

Accessing the newly created project from the UART_led.vhd file:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

library xil_defaultlib;
use xil_defaultlib.UART_led_pkg.all;

library UNISIM;
use UNISIM.VComponents.all;
```


© Copyright 2019 Xilinx

Lab 9: Building a Moore Finite State Machine

2019.1

Abstract

This lab illustrates how to build a basic Moore Finite State Machine (FSM) using a single-process state machine.

This lab should take approximately 60 minutes.

Objectives

After completing this lab, you will be able to:

- Construct a single-process Moore-style FSM

Introduction

The module that you will build in this lab will control the output of an RS-232 serial transmitter that can be included as part of a testbench for providing data to a UART receiver or as part of a synthesizable design.

The signals are fairly straightforward:

- Data to transmit signal which arrives as an 8-bit parallel byte
- Begin transmission signal when asserted, causes the state machine to begin the transmission sequence
- Baud rate enable signal that is high for one clock cycle and is produced at the desired baud rate
- Reset signal that immediately causes the state machine to return to IDLE state and drive the serial output to line idle
- Operational clock signal
- Serialized output signal

© Copyright 2019 Xilinx

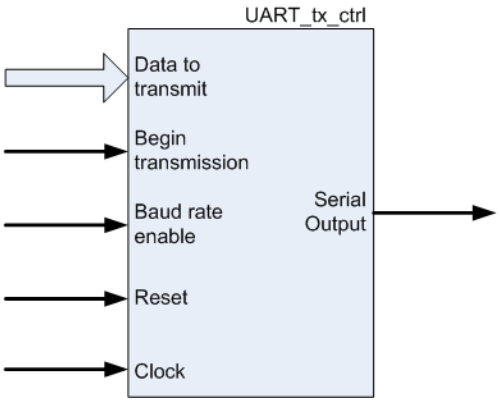


Figure 9-1: I/O Signals for UART_tx_ctrl

To properly build the state machine, you must understand the waveform that is to be produced.

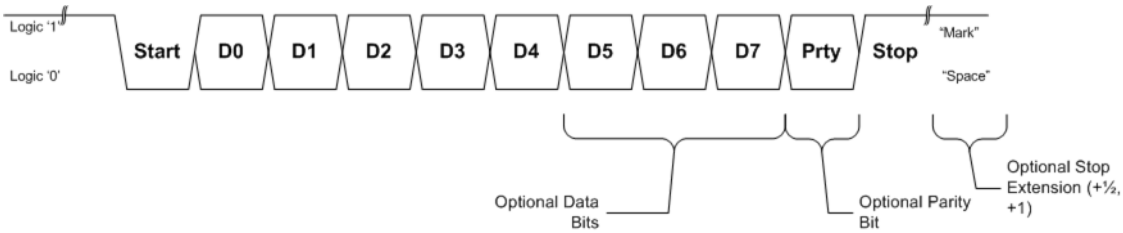


Figure 9-2: RS-232 Waveform

This particular design specifies eight data bits with no parity and no stop bits.

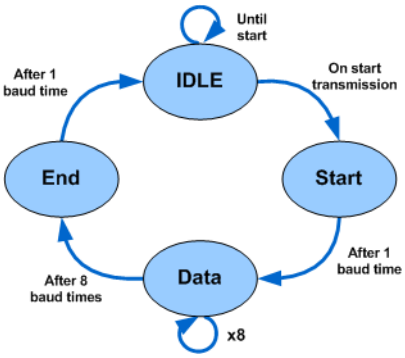


Figure 9-3: FSM Diagram

The above FSM diagram is a very simple state diagram, and performance is unlikely to be an issue (considering that the baud rate is only 115,200), you will build this as a single-process state machine.

Notice that the state machine resides in the data state for eight baud times. This implies a counter. The counter could be "built in" to the state machine via extra states. Instead of one DATA state that monitors a counter outside the state machine, there could be eight data states, each one responsible for generating one of the bits.

© Copyright 2019 Xilinx

It is important to note that although it is possible to build the counter into the state machine, the performance of the FSM will be considerably lower. This would be acceptable for this FSM; however, you will follow best design practice and keep the counter external to the FSM.

Understanding the Lab Environment

The labs and demos provided in this course are designed to run under the provided Linux platform. The provided Linux platform has a customized lab environment. These customizations have been made to simplify and enhance your learning experience. Many labs and demos can be successfully executed in the Windows environment as well. Topics including PetaLinux, Yocto, and QEMU are generally not supported under Windows.

The instructions, for the sake of brevity, are expressed using the Linux notation. This includes the forward slash ('/') as the hierarchy separator instead of the Windows backslash ('\'). For students who want to run under Windows, it is their responsibility to use the correct hierarchy separator.

Windows users can run Linux using the VirtualBox tool from Oracle. Details on how to acquire and install this tool can be found in the lab setup guide. The Xilinx Customer Education group offers a preconfigured virtual machine image and includes all the tools necessary to run the labs. Some tools and IP do require licenses, which are the responsibility of the student.

Several environment variables, described in detail in the lab setup guide, which also discusses how you can customize the environment for your situation, are provided for ease of use. Most notably, many environment variables have been defined to reduce the amount of typing you need to do when entering paths.

These environment variables are:

- XILINX_PATH - points to the installation directory for the Xilinx tools. Tools such as the Vivado Design Suite and SDK can be found here.
- PETALINUX_PATH - points to the installation directory for the Xilinx PetaLinux tools. Often this is within XILINX_PATH, but some choose to install it elsewhere.
- TRAINING_PATH - points to the space allocated for students to work through their labs. This directory includes prebuilt images and starting points for the labs and demos.
- Moore_FSM - points to the specific name of the topic within the TRAINING_PATH directory.

  This lab uses $Moore_FSM to indicate the directory for this lab, which is equivalent to $TRAINING_PATH/Moore_FSM.

These environment variables work well in the Linux terminal and the Vivado Design Suite and scripts, but not in Eclipse-based tools such as the SDK tool and the SDx™ environment. When using the Vivado tools, you can use $Moore_FSM and the tools will automatically expand the environment variable for you.

However, when you see the notation $Moore_FSM/workspace in SDK or the SDx environment, you should interpret it in its expanded form. The expanded form can be found by opening a terminal window and enter "echo $Moore_FSM".

© Copyright 2019 Xilinx

This lab assumes that you are starting with a Linux environment. If you are working on a Windows system, refer to the lab setup guide to see how to use the Oracle VirtualBox application to run the Xilinx Customer Education pre-configured Ubuntu Linux virtual machine.

General Flow

Creating a New Project — Step 1

1-1. Launch the Vivado Design Suite.

If you do not recall how to perform this task, refer to the "Launching the Vivado Design Suite" section under Vivado Design Suite Operations in the *Lab Reference Guide*.

1-2. Create a new Vivado Design Suite project named *Moore_FSM* and locate it in the following directory.

Browse to the `$Moore_FSM/lab/KCU105` directory.

Target the project for the Kintex UltraScale KCU105 Evaluation Platform evaluation board.

If you do not recall how to perform this task, refer to the "Creating a Blank Vivado Design Suite Project" section in the *Lab Reference Guide*.

From the settings in the Flow Navigator, change the target language to VHDL.

© Copyright 2019 Xilinx

Creating a New VHDL Source Step 2

Now that the project has been created, it is time to begin creating the design. You will begin by creating a VHDL module template, then filling the template with two processes: a Moore FSM and a supporting counter.

The care that needs to be taken is in the Moore FSM, there are outputs that depend upon only the current states.

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

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

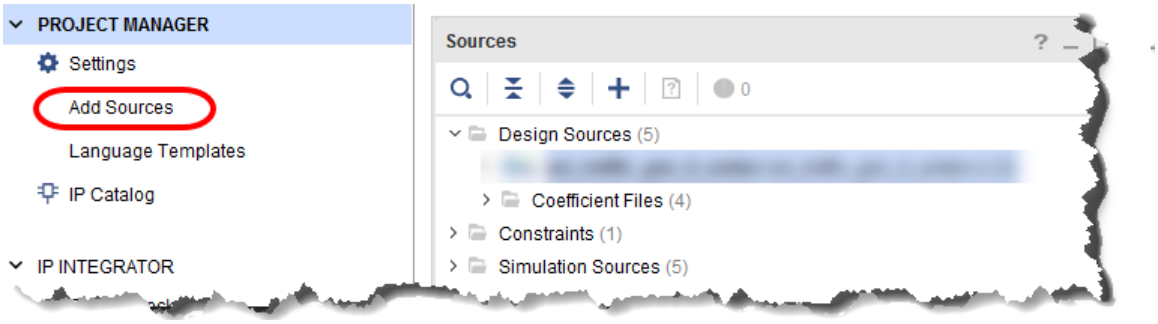


Figure 9-4: Selecting Add Sources

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

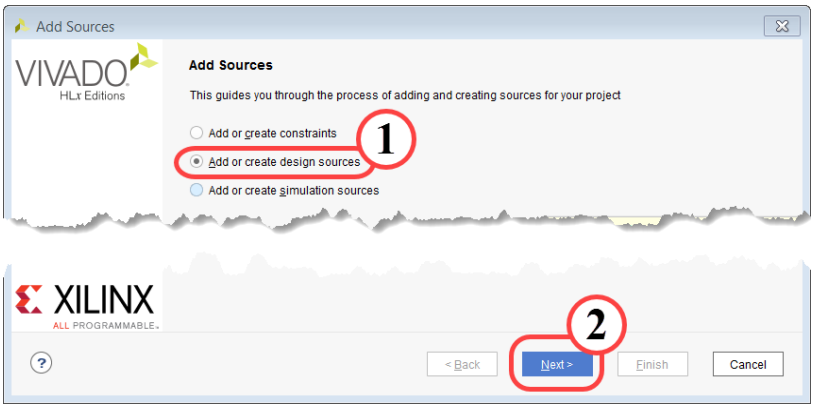


Figure 9-5: Selecting Add or Create Design Sources

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

The Add or Create Design Sources dialog box opens.

© Copyright 2019 Xilinx

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

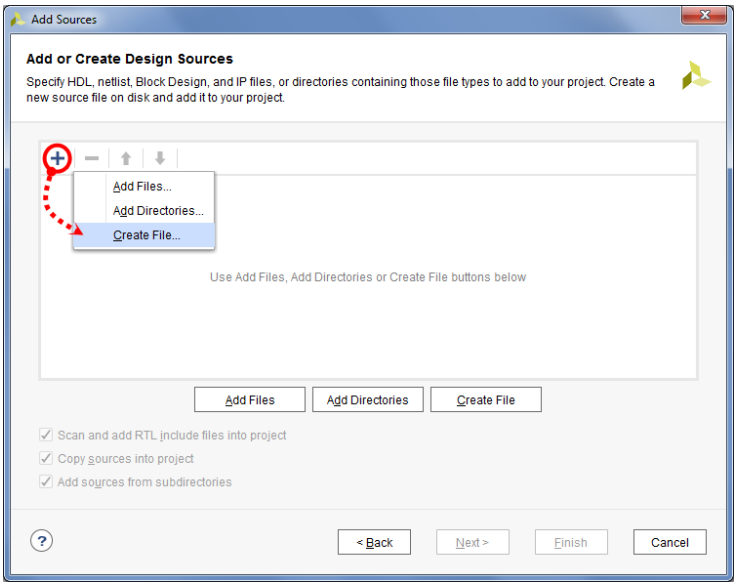


Figure 9-6: Selecting Create File

The Create Source File dialog box opens.

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

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

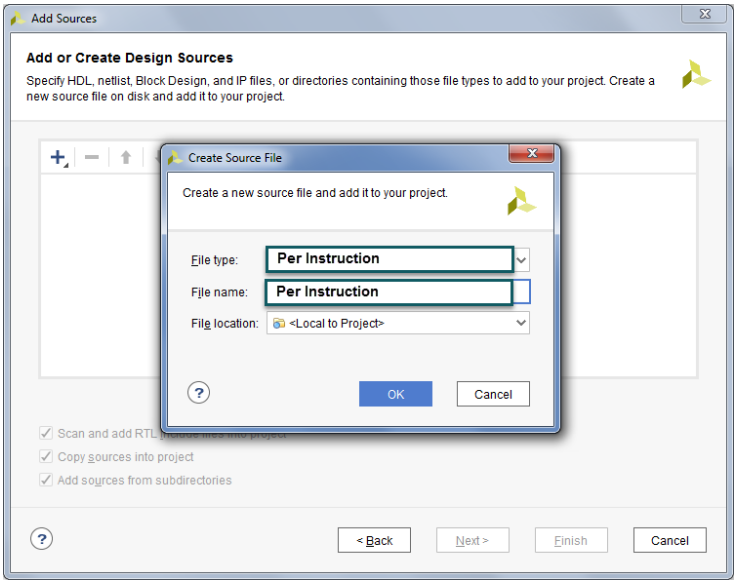


Figure 9-7: Entering File Name and Type

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.

© Copyright 2019 Xilinx

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

- **tx_data_in – in - std_logic_vector – width 8**
- **tx_start – in - std_logic**
- **tx_baud_rate_enable – in – std_logic**
- **tx_serial_out – out – std_logic**
- **clock – in – std_logic**
- **reset – in – std_logic**

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

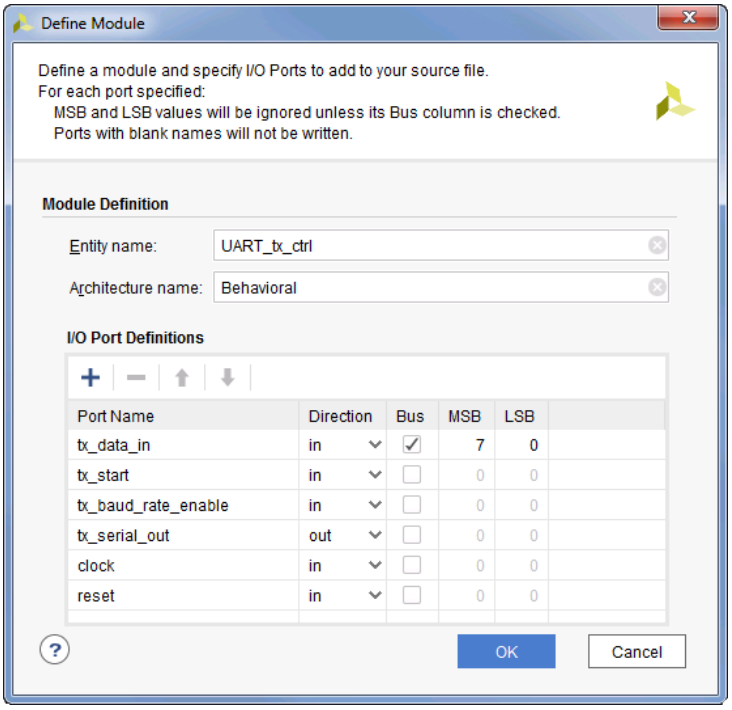


Figure 9-8: Defining Ports for UART_tx_ctrl

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

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

2-3-1. Double-click the **UART_tx_ctrl.vhd** tab to make the source code visible in the main workspace window.

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

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

© Copyright 2019 Xilinx

2-4. Now that you have defined the inputs and outputs for this module, the behavior must be described. There are three basic functions for this module: a one-shot process that can be retriggered for capturing the start request; the state machine itself; and a counter for counting the number of bits transmitted.

Begin with coding the down counter.

Using the Language Templates, select the snippet for a simple down counter with enable and synchronous reset.

2-4-1. Select **Tools** > **Language Templates** to open the Language Templates.

Alternatively, the Language Templates option is available in the Flow Navigator.

The Language Templates contain a large number of code snippets that demonstrate how various constructs can be built.

2-4-2. Expand **VHDL** > **Synthesis Constructs** > **Coding Examples** > **Counters** > **Binary** > **Down Counters**. Select **/w CE and Sync Active High Reset**.

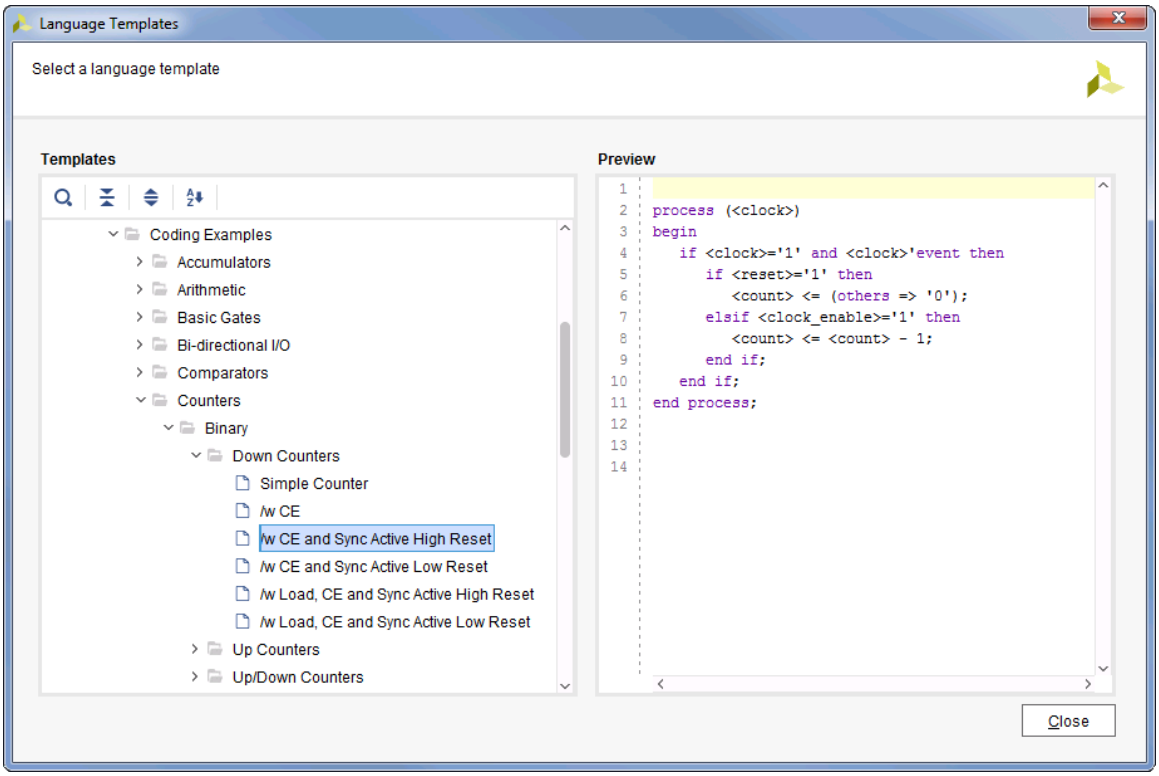


Figure 9-9: Partially Expanded Language Template List

© Copyright 2019 Xilinx

2-5. Copy and paste the entire snippet into the source code between the "begin" and "end Behavioral" statements. Indent the snippet. Remember to indent the process.

2-5-1. Select the text in the Preview window of the Language Templates tab.

2-5-2. Right-click the highlighted area and select **Copy**.

2-5-3. Click the **UART_tx_ctrl** tab to switch back to the source code.

The asterisk in the tab means that this file is unsaved.

2-5-4. Click anywhere between the "begin" and "end Behavioral" statements.

2-5-5. Right-click and select **Paste**. You can also use the keyboard shortcut <**Ctrl** + **V**>.

Good design practice calls for intelligent use of white space.

2-6. The snippet represents generic code.

Change the items enclosed in the angle brackets with signals that were defined in the entity.

2-6-1. Make the following substitutions:

- <clock> becomes clock
- <reset> becomes reset
- <clock_enable> becomes tx_baud_rate_enable
- <count> becomes count_value
- To improve readability, replace the "<clock>'event and <clock>='1'" with rising_edge(clock).
- Because the intent is to count down from 7 to 0 and use this value as the index into the tx_data_in, replace (others=> '0') to 7;

The code should appear as in the figure below.

```
process (clock)
begin
   if rising_edge(clock) then
      if reset='1' then
       count_value <= 7;
      elsif tx_baud_rate_enable='1' then
         count_value <= count_value - 1;
      end if;
   end if;
end process;
```

Figure 9-10: Snippet with Substitutions

2-6-2. Modify the reset condition as follows to allow the counter to be reset by both the FSM and a signal external to the module.

if (reset='1') or (count_reset='1') then

© Copyright 2019 Xilinx

2-6-3. Make the appropriate additions to the architecture declaration area to account for the two new signals you have introduced (count_value and count_reset).

```
43  architecture Behavioral of UART_tx_ctrl is
44
45  signal count_value  :   integer range 0 to 7 := 7;
46  signal count_reset  :   std_logic := '0';
47
48  begin
```

Figure 9-11: Counter Signals Defined

Notice that count_value is an integer. This simplifies use considerably. Because this value will be used as an index into an array (std_logic_vector), you will not need to perform a conversion on it.

2-7. Construct a one-shot process that can be retriggered.

This is a commonly used construct that is used to detect a small or short-lived event and hold it until a slower task can use the information.

The tx_start signal can only be one clock cycle long but the state machine that you will be constructing next is only active at the baud rate 115200. Thus, the probability of catching this event is 10 ns out of 8.681 µs or 1 out of 868! Obviously this is unacceptable as the pulse must be caught each and every time.

Use the following signals to construct one-shot process:

- **start_transmitting – std_logic – connects the fired output of the one-shot to the FSM**
- **reset_oneShot – std_logic – FSM control to clear the one-shot and prepare for the next event**
- **registered_data – std_logic_vector(7 downto 0) – stores the data to transmit when the tx_start is asserted**

Although suggested code solutions can be found in the Answers section, you should make every effort to code these yourself.

2-7-1. Define the signals listed above in the architecture's declaration area.

2-7-2. Write the outline of a process named oneShot.

This process should be synchronous with a synchronous reset.

2-7-3. Add code that will pull start_transmitting high when tx_start is asserted as well as capture the character to send which is presented on tx_data_in.

2-7-4. Add a mechanism for clearing the one-shot so that the FSM can reset the device for future tx_start events.

© Copyright 2019 Xilinx

Question 1

Is this one-shot process complete? If not, what is missing?

_____

_____

_____

2-8. You can finally start coding the FSM. Begin by creating a synchronous process with synchronous reset (similar to the outline of the one-shot process that you previously wrote) but named FSM_tx. Define the states necessary to implement the four states shown in the state diagram.

Although a suggested code solution can be found in the Answers section, you should make every effort to code these yourself.

2-8-1. Write the outline of a process named FSM_tx.

This process should be synchronous with a synchronous reset.

2-8-2. Define the four states IDLE, START, DATA, and STOP as an enumerated type named valid_states. Create a variable name state which is of the valid_states type.

2-8-3. Add the test to this process, since it can only operate when the tx_baud_rate_enable is active.

2-8-4. Create the four states as a case statement and create a space within each state for activities associated with that state and transitions associated with that state.

2-8-5. Fill out the IDLE state.

This state must:

- Hold the output at line idle, which for RS-232 is logic high.
- Hold the counter in reset to keep it pre-loaded and not counting.
- Enable the one-shot to accept any new tx_start requests.
- Transition to the START state when the output of the one-shot is active (start_transmitting).

2-8-6. Repeat the above process for the START state.

This state must:

- Drive the output to low, which is the RS-232 start bit condition.
- Release the counter to begin the count for sending data.
- Transition to the DATA state on the next baud enable.

© Copyright 2019 Xilinx

2-8-7. Repeat for the DATA state using the process described above.

This state must:

- Drive the output based on the value indexed by count_value.
- Reset the counter when the count is complete.
- Transition to the STOP state when the count is complete.

2-8-8. Code the STOP state.

This state must:

- Drive the serial output to logic high (for the stop bit).
- Clear the one-shot in preparation for the next event.
- Always transition back to IDLE on the next baud rate.

2-8-9. Code the reset portion.

Because this process is responsible for driving the reset_oneShot, count_reset, and tx_serial_out, define the reset values for these signals in the proper location within the FSM_tx process. The initial state should also be specified here.

2-8-10. Select **File** > **Text Editor** > **Save File**.

Simulating the Design — Step 3

This portion of this lab requires you to load a predefined configuration for the simulation environment, run the simulation, and view the waveform.

3-1. Add the provided UART_tx_ctrl_tb.vhd testbench to the design and note the order and appearance of the signals.

3-1-1. Click **Add Sources** under Project Manager in the Flow Navigator.

The Add Sources Wizard opens.

3-1-2. Select **Add or create simulation sources** and click **Next**.

3-1-3. Click the **Plus** (+) icon and select **Add Files**.

3-1-4. Select **UART_tx_ctrl_tb.vhd** and click **OK** and click **Finish**.

3-2. Run behavioral simulation using the Vivado simulator.

3-2-1. Select **Simulation** > **Run Simulation** > **Run Behavioral Simulation** from the Flow Navigator under Simulation.

The simulation window opens and simulation runs for the length of time specified in the Simulation Settings (10 us default).

© Copyright 2019 Xilinx

You can alternatively start the the Vivado simulator by entering `launch_simulation` in the Tcl command line.

3-3. Load the provided configuration file FSM.wcfg and set the run time for 120 µs.

3-3-1. Select **File** > **Simulation Waveform** > **Open Configuration**.

3-3-2. Select **FSM.wcfg**.

3-3-3. Click **OK** in the Open Waveform Configuration dialog box.

3-3-4. Select **Run** > **Restart** (or click the icon) to reset the simulation.

3-3-5. Select **Run** > **Run For** to run the simulation for the specified time.

The **Run For** dialog box opens and prompts you to specify the amount of time to run the simulation for and time units.

3-3-6. Enter 120 us in the Time field.

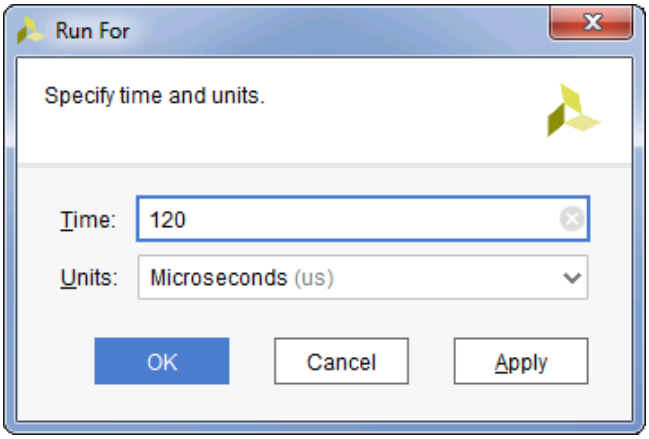


Figure 9-12: Run For Dialog Box

3-3-7. Click **OK**.

3-4. Examine the waveform, review the testbench, and answer the questions below.

Question 2

When does the request to transmit signal occur (tx_start)?

_____

_____

_____

© Copyright 2019 Xilinx

Question 3

How long after tx_start is asserted does the transmission actually begin?

_____

_____

_____

Question 4

Is the design working?

_____

_____

_____

3-5. Exit the Vivado simulator.

3-5-1. Select **File** > **Close Simulation**.

3-6. Close the project.

3-7. Close Vivado Design Suite.

[Optional]: Clean up the file system.

Some systems 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. Both processes will recursively delete all of the files in the `$Moore_FSM` directory.

3-8. Clean up the VirtualBox file system via the GUI.

3-8-1. Using the graphical browser (Windows: press the <**Windows**> key + <**E**>; Linux: press <**Ctrl** + **N**>), navigate to `$Moore_FSM`.

3-8-2. Select **`Moore_FSM`**.

3-8-3. Press <**Delete**>.

© Copyright 2019 Xilinx

3-9. Clean up the VirtualBox file system via the command line.

3-9-1. Open a terminal window (Windows: press the <**Windows**> key + <**R**>, then enter **cmd**; Linux: press <**Ctrl** + **Alt** + **T**>)

3-9-2. Enter the following command to delete the contents of the workspace:

[Windows users]: `rd /s /q $Moore_FSM`

[Linux users]: `rm -rf $Moore_FSM`

Summary

This lab demonstrated how to construct a simple, but viable, one-process Moore state machine where the outputs depend upon only the current states. You coded two common constructs: that of a one-shot, and the other of a simple counter.

As a simple state machine, this single-process solution is adequate for mid-to-low performance tasks. As the state machine becomes larger or the states become more complex, using a multi-process FSM is recommended.

© Copyright 2019 Xilinx

Answers

Answers listed represent sample solutions only. Your results may differ depending on the version of the software, service pack, or operating system that you are using.

1. Is this one-shot process complete? If not, what is missing?

   The design of this one-shot process makes an assumption that the request to start is removed prior to the one-shot being cleared. As the design stands now, if the request to start remains asserted, whatever data is presented to the transmitter when the one-shot is cleared will be retransmitted.

   A superior design is to prevent the one-shot from triggering unless the request has been deasserted.

   Consider adding the following code to the triggering mechanism:

   This requires two more lines of code to be added: one that defines last_tx_start as a std_logic variable within the process and the other that updates this new variable each clock cycle. In this fashion, the one-shot can only be retriggered on the rising edge of tx_start.

   Use the following figure to answer the questions from Question 2 to Question 4.

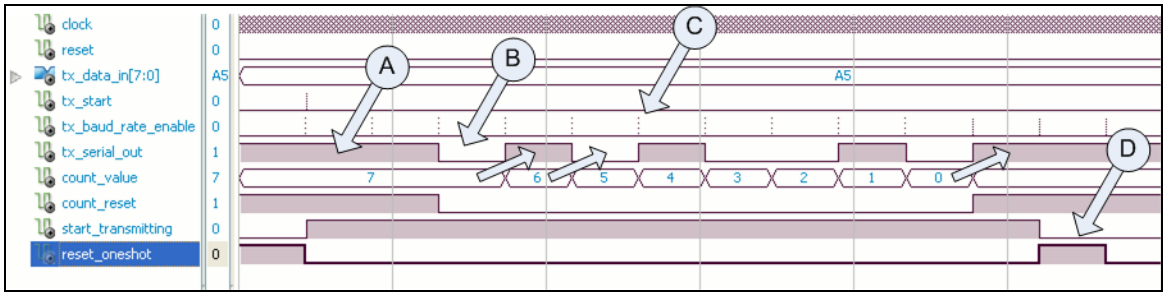


Figure 9-13: Waveform for Questions 2–4

2. When does the request to transmit signal occur (tx_start)?

   tx_start is asserted for one clock cycle at 8,870.550 ns.

3. How long after tx_start is asserted does the transmission actually begin?

   The start bit drops low after approximately 10 µs. This is due to the start request occurring slightly after a baud rate enable pulse. In this situation, the FSM had to wait until the next baud rate enable occurred to begin operating the state machine.

   A slightly more efficient FSM would remove the START state and put its contents into the if block that controlled the transition to START. In this way, the message would be sent one baud rate time sooner.

© Copyright 2019 Xilinx

4. Is the design working?

   Brief analysis:

   Notice that the serial line is held high (‘A’) until the stop bit is transmitted at ‘B’.

   The transmitted data is hexadecimal A5 or 10100101 in binary. This makes both the most significant bit and least significant bit a ‘1’. This is a poor choice for a test pattern as it is symmetric. A better choice could have been E1 hex.

   Looking at the ordering of the data compared with the RS-232 protocol diagram in Figure 6-2, the data is being transmitted in the reverse order. Clearly this needs to be fixed.

   This can be fixed by rebuilding the counter so that it is a count-up device instead of a count-down device. This will require that the initial values and limit test be updated.

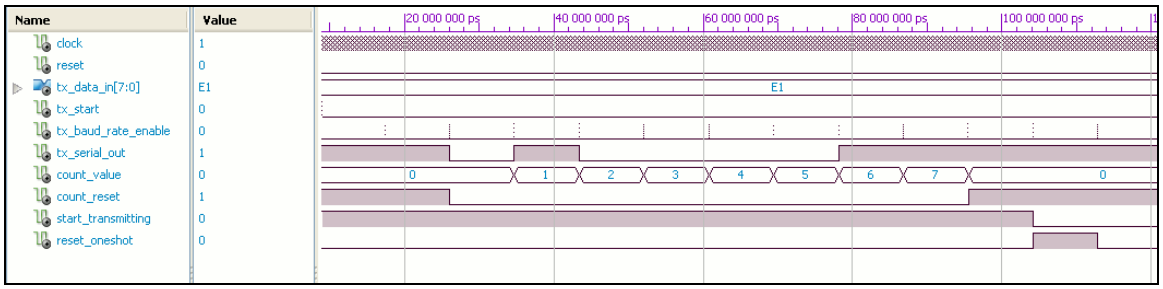


Figure 9-14: Fixed Waveform

© Copyright 2019 Xilinx

OneShot Coding Process

Code for the oneShot coding process:

```
-- one-shot to capture short duration events
-- in this case, the tx_start
oneShot: process (clock)
    begin
        if rising_edge(clock) then                  -- active only on clock edge
            if (reset ='1') or                      -- is external reset asserted?
               (reset_oneShot = '1') then           -- what about a reset from the FSM?
                start_transmitting <= '0';          -- don't tell the FSM to begin
            else                                            -- reset not asserted
                if (tx_start = '1') then            -- has a start been requested?
                    start_transmitting <= '1';      -- let the FSM know
                    registered_data <= tx_data_in;-- store the data while valid
                end if;                                 -- end of tx_start test
            end if;                                     -- end of reset/non-reset activities
        end if;                                         -- end of synch events
    end process oneShot;
```

Coding the FSM

Outline of a process named FSM_tx:

```
-- Finite State Machine for controlling serial tramsission
FSM_tx: process (clock)
   begin
      if rising_edge(clock) then              -- active only on clock edge
         if (reset = '1') then                -- is the external reset asserted?
         else                                 -- reset not asserted
         end if;                              -- end of reset/non-reset activities
      end if;                                 -- end of synch activities
   end process FSM_tx;
```

Four states IDLE, START, DATA, and STOP as an enumerated type named valid_states. Variable name state which is of the valid_states type:

```
-- Finite State Machine for controlling serial tramsission
FSM_tx: process (clock)
      type valid_states is (IDLE, START, DATA, STOP);
      variable state           : validStates := IDLE;
   begin
```

Test for this process:

```
-- Finite State Machine for controlling serial tramsission
FSM_tx: process (clock)
      type valid_states is (IDLE, START, DATA, STOP);
      variable state  : valid_states := IDLE;
   begin
      if rising_edge(clock) then              -- active only on clock edge
         if (reset = '1') then                -- is the external reset asserted?
         else                                 -- reset not asserted
            if (tx_baud_rate_enable = '1') then -- don't operate every clock, only @ baud

            end if;                           -- end of baud enable test
         end if;                              -- end of reset/non-reset activities
      end if;                                 -- end of synch activities
   end process FSM_tx;
```

Note: This could also have been coded using "elsif (tx_baud_rate_enable = ‘1’) then …"

© Copyright 2019 Xilinx

Four states as a case statement and space within each state for activities associated with that state and transitions associated with that state:

```
-- Finite State Machine for controlling serial tramsission
FSM_tx: process (clock)
      type valid_states is (IDLE, START, DATA, STOP);
      variable state           : validStates := IDLE;
   begin
      if rising_edge(clock) then              -- active only on clock edge
         if (reset = '1') then                -- is the external reset asserted?
         else                                 -- reset not asserted
            case state is                     -- decide which state to "do"
               when IDLE =>
                  -- actions
                  -- transitions
               when START =>
                  -- actions
                  -- transitions
               when DATA =>
                  -- actions
                  -- transitions
               when STOP =>
                  -- actions
                  -- transitions
            end case;                         -- end of states
         end if;                              -- end of reset/non-reset activities
      end if;                                 -- end of synch activities
   end process FSM_tx;
```

IDLE state:

```
when IDLE =>
      -- actions
      tx_serial_out <= '1';   -- idle the RS-232 line
      reset_oneShot <= '0';   -- enable the one-shot to look for xmit requests
      count_reset   <= '1';   -- preload the counter and don't start counting

      -- transitions
      if (start_transmitting = '1') then  -- transition when request is made
         state := START;      -- to the START state
      end if;
```

START state:

```
when START =>
      -- actions
      tx_serial_out <= '0';       -- start bit is always 0
      count_reset <= '0';         -- let the counter count

      -- transitions
      state := DATA;              -- always happens
```

DATA state:

```
when DATA =>
   -- actions
   tx_serial_out <= registered_data(count_value);  -- select the proper bit
   if (count_value = 0) then
      count_reset <= '1';
   end if;

   -- transitions
   if (count_value = 0) then  -- if done
      state := STOP;
   end if;
```

Note: This could also have been coded with both the count reset and next state within the same if block.

© Copyright 2019 Xilinx

STOP state:

```
when STOP =>
   -- actions
   tx_serial_out <= '1';      -- stop bit is always line idle
   reset_oneShot <= '1';      -- clear the 1shot for next event

   -- transitions
   state := IDLE;             -- always
```

Reset portion:

```
if rising_edge(clock) then              -- active only on clock edge
   if (reset = '1') then                -- is the external reset asserted?
      state := IDLE;
      tx_serial_out <= '1';             -- line idle
      reset_oneShot <= '1';             -- keep it reset
      count_reset   <= '1';             -- preload the counter and prevent from counting

   else                                 -- reset not asserted
```

Final Code

The final code also includes the correction per Question 4, which is why the counter is an UP counter rather than a DOWN counter.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity UART_tx_ctrl is
    Port ( tx_data_in : in STD_LOGIC_VECTOR (7 downto 0);
           tx_start : in STD_LOGIC;
           tx_baud_rate_enable : in STD_LOGIC;
           tx_serial_out : out STD_LOGIC;
           clock : in STD_LOGIC;
           reset : in STD_LOGIC);
end UART_tx_ctrl;

architecture Behavioral of UART_tx_ctrl is

    signal count_value    :    integer range 0 to 7 := 7;
    signal count_reset    :    std_logic := '0';
    -- signals associated with the one-shot
    signal start_transmitting     : std_logic := '0';
```


© Copyright 2019 Xilinx

```
    signal reset_oneShot            : std_logic := '0';
    signal registered_data          : std_logic_vector(7 downto 0) := (others=>'0');

begin
    -- count up from 0 to 7
    process (clock)
    begin
        if rising_edge(clock) then
            if (reset='1') or (count_reset='1') then
                count_value <= 0;
            elsif tx_baud_rate_enable='1' then
                count_value <= count_value + 1;
            end if;
        end if;
    end process;

    -- one-shot to capture short duration events
    -- in this case, the tx_start
    oneShot: process (clock)
    begin
        if rising_edge(clock) then                  -- active only on clock edge
            if (reset ='1') or                              -- is external reset asserted?
                (reset_oneShot = '1') then                  -- what about a reset from the FSM?
                start_transmitting <= '0';          -- don't tell the FSM to begin
            else                                    -- reset not asserted
                if (tx_start = '1') then                    -- has a start been requested?
                    start_transmitting <= '1';              -- let the FSM know
                    registered_data <= tx_data_in;-- store the data while valid
                end if;                             -- end of tx_start test
            end if;                                 -- end of reset/non-reset activities
        end if;                                     -- end of synch events
```


© Copyright 2019 Xilinx

```
    end process oneShot;

    -- Finite State Machine for controlling serial transmission
    FSM_tx: process (clock)
      type valid_states is (IDLE, START, DATA, STOP);
      variable state    : valid_states := IDLE;
      begin
        sync_events: if rising_edge(clock) then          -- active only on clock edge
            reset_event:if (reset = '1') then                -- is the external reset asserted?
                state := IDLE;                                  -- start in the IDLE state
                tx_serial_out <= '1';                           -- line idle
                reset_oneShot <= '1';                           -- keep it reset
                count_reset      <= '1';                        -- preload the counter and prevent
from counting
            else                                                          -- reset not asserted
                baud_enable: if (tx_baud_rate_enable = '1') then    -- don't operate every
clock, only @ baud
                    state_machine: case state is
                        when IDLE =>
                            -- actions
                            tx_serial_out <= '1';        -- idle the RS-232 line
                            reset_oneShot <= '0';      -- enable the one-shot to look for
xmit requests
                            count_reset      <= '1';     -- preload the counter and don't
start counting

                            -- transitions
                            if (start_transmitting = '1') then   -- transition when request is
made
                                state := START;                          -- to the START state
                            end if;

                        when START =>
```


© Copyright 2019 Xilinx

```
                    -- actions
                    tx_serial_out <= '0';              -- start bit is always 0
                    count_reset <= '0';                     -- let the counter
count

                    -- transitions
                    state := DATA;                          -- always happens

                when DATA =>
                    -- actions
                    tx_serial_out <= registered_data(count_value);  -- select the
proper bit
                    if (count_value = 7) then
                        count_reset <= '1';
                    end if;

                    -- transitions
                    done_test: if (count_value = 7) then
                        state := STOP;
                    end if done_test;

                when STOP =>
                    -- actions
                    tx_serial_out <= '1';              -- stop bit is always line idle
                    reset_oneShot <= '1';              -- clear the 1shot for next
event

                    -- transitions
                    state := IDLE;                     -- always

            end case state_machine;
        end if baud_enable;
```


© Copyright 2019 Xilinx

```
            end if reset_event;
        end if sync_events;
    end process FSM_tx;

end Behavioral;
```


© Copyright 2019 Xilinx

Lab 10: Building a Meaningful Testbench

2019.1

Abstract

This lab demonstrates one mechanism for creating a self-checking, time-agnostic testbench. This lab does not require you to trace through the waveforms to validate the design. Instead the self-checking testbench will trace the waveform and report if any issues are found.

This lab should take approximately 60 minutes.

Objectives

After completing this lab, you will be able to:

- Illustrate basic techniques used to assemble a self-checking, time-agnostic testbench

Introduction

The testbench that you will build in this lab is more sophisticated and is intended for verification of major sub-components within a larger design and the top-level design itself. This level of construction is *not* required to validate low-level modules to check basic function.

The term self-checking indicates that the waveform contains useful information for debugging, but the code will provide a clear indication of the test being passed or not, as well as any relevant messages. Therefore, this type of set up is used, and the simulation only needs to be run for "long enough" (based on the test criteria) and the simulation will stop itself and report which aspects of the test passed and which failed. In this way, the results can easily be included in validation documentation.

The time-agnostic aspect of this testbench is derived from the fact that regardless of the type of simulation being run (behavioral, post-synthesis, or post-implementation), the testbench will always locate the proper piece of information coming out of the UUT and compare it with a known good datum, irrespective of any delays due to simulation method.

You will implement the block diagram shown in the figure below.

The data generator produces a series of characters that are sent to the FIFO (which manages the time through the UUT) and the serializer which creates a proper RS-232 bitstream for the UUT.

The comparator compares the output of the FIFO with the pattern produced on led_pins. Because there is no led_pins_valid signal, a data change detector is necessary to indicate to the comparator that the next set of data is available for checking.

© Copyright 2019 Xilinx

This means that the test data used by the data generator cannot have two of the same characters in a row; otherwise, the comparator will not know that the second datum is new and will cause the test to fail.

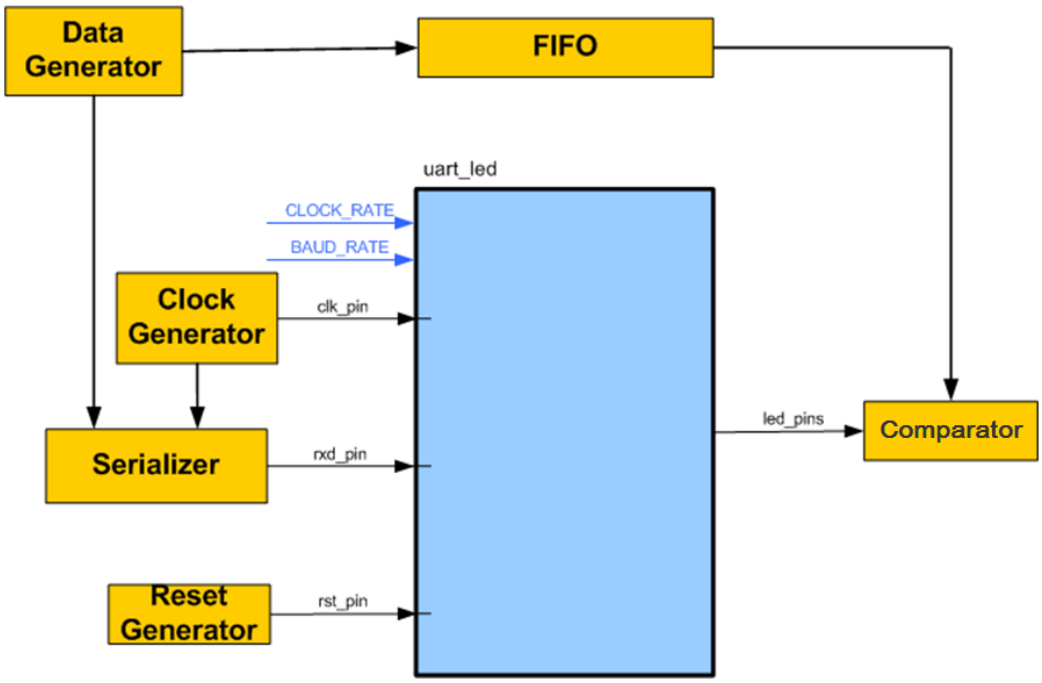


Figure 10-1: Block Diagram for Self-Checking Testbench

Understanding the Lab Environment

The labs and demos provided in this course are designed to run under the provided Linux platform. The provided Linux platform has a customized lab environment. These customizations have been made to simplify and enhance your learning experience. Many labs and demos can be successfully executed in the Windows environment as well. Topics including PetaLinux, Yocto, and QEMU are generally not supported under Windows.

The instructions, for the sake of brevity, are expressed using the Linux notation. This includes the forward slash ('/') as the hierarchy separator instead of the Windows backslash ('\'). For students who want to run under Windows, it is their responsibility to use the correct hierarchy separator.

Windows users can run Linux using the VirtualBox tool from Oracle. Details on how to acquire and install this tool can be found in the lab setup guide. The Xilinx Customer Education group offers a preconfigured virtual machine image and includes all the tools necessary to run the labs. Some tools and IP do require licenses, which are the responsibility of the student.

Several environment variables, described in detail in the lab setup guide, which also discusses how you can customize the environment for your situation, are provided for ease of use. Most notably, many environment variables have been defined to reduce the amount of typing you need to do when entering paths.

© Copyright 2019 Xilinx

These environment variables are:

- XILINX_PATH - points to the installation directory for the Xilinx tools. Tools such as the Vivado Design Suite and SDK can be found here.
- PETALINUX_PATH - points to the installation directory for the Xilinx PetaLinux tools. Often this is within XILINX_PATH, but some choose to install it elsewhere.
- TRAINING_PATH - points to the space allocated for students to work through their labs. This directory includes prebuilt images and starting points for the labs and demos.
- selfChecker - points to the specific name of the topic within the TRAINING_PATH directory.

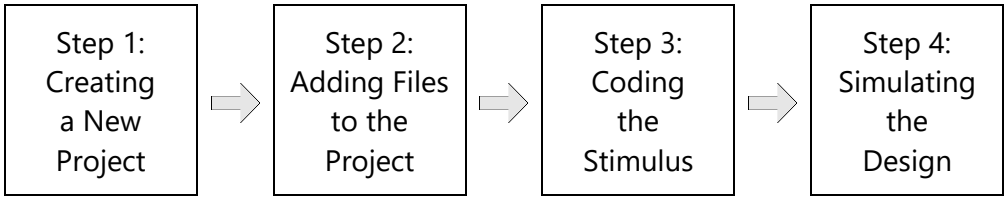
  This lab uses $selfChecker to indicate the directory for this lab, which is equivalent to $TRAINING_PATH/selfChecker.

These environment variables work well in the Linux terminal and the Vivado Design Suite and scripts, but not in Eclipse-based tools such as the SDK tool and the SDx™ environment. When using the Vivado tools, you can use $selfChecker and the tools will automatically expand the environment variable for you.

However, when you see the notation $selfChecker/workspace in SDK or the SDx environment, you should interpret it in its expanded form. The expanded form can be found by opening a terminal window and enter "echo $selfChecker".

This lab assumes that you are starting with a Linux environment. If you are working on a Windows system, refer to the lab setup guide to see how to use the Oracle VirtualBox application to run the Xilinx Customer Education pre-configured Ubuntu Linux virtual machine.

General Flow



© Copyright 2019 Xilinx

Creating a New Project — Step 1

1-1. Launch the Vivado Design Suite.

If you do not recall how to perform this task, refer to the "Launching the Vivado Design Suite" section under Vivado Design Suite Operations in the *Lab Reference Guide*.

1-2. Create a new Vivado Design Suite project named *selfChecker* and locate it in the following directory.

Browse to the `$selfChecker/lab/KCU105` directory.

Target the project for the Kintex UltraScale KCU105 Evaluation Platform evaluation board.

If you do not recall how to perform this task, refer to the "Creating a Blank Vivado Design Suite Project" section in the *Lab Reference Guide*.

From the settings in the Flow Navigator, change the target language to VHDL.

Adding Files to the Project — Step 2

A number of files have been provided for you, including a number of testbench-supporting files and all of the source files for the project.

2-1. You will load the following source files found in the working directory into the project.

- **uart_led.vhd – top level of the design**
- **meta_harden – anti-meta-stable circuit**
- **uart_rx.vhd – UART receiver**
- **uart_baud_gen.vhd – baud rate generator for oversampling serial data**
- **uart_rx_ctl.vhd – UART state machine**
- **register8.vhd – 8-bit registering flip-flop module**
- **LED_manager.vhd – multiplexer to select functionality**
- **reset_bridge.vhd – for safely applying reset**

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

The Add Sources Wizard opens.

© Copyright 2019 Xilinx

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

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

2-1-4. Click the **Plus** (+) icon and select **Add Files**.

2-1-5. Browse to the `$selfChecker/lab/KCU105` directory if it is not open already.

2-1-6. Press the <**Ctrl**> key and select all the files listed above.

2-1-7. Click **OK**.

2-1-8. Ensure that the **Copy sources into project** option is selected.

2-1-9. Click **Finish** to confirm adding the selected files to the project with the default associations and libraries.

2-2. A number of packages have been included to make the I/O portion of simulation flow more easily. These packages can be used in other testbenches as needed.

Add the following packages to the appropriate libraries:

- **string_utilities_sim_pkg to utilities_lib**
- **string_utilities_synth_pkg to utilities_lib**
- **time_utilities_pkg to utilities_lib**
- **fifo_pkg to specific_support_lib**

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

The Add Sources Wizard opens.

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

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

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

2-2-5. Press the <**Ctrl**> key and select the following files:

- *fifo_pkg.vhd*
- *string_utilites_sim_pkg.vhd*
- *string_utilities_synth_pkg.vhd*
- *time_utilities_pkg.vhd*

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

Although it is possible to add the packages to the *xil_defaultlib* library, you will add them to two new VHDL libraries. This is usually done to keep the sources separate. In general, unless you are actively developing a package, these sources will not need to be changed. To help avoid accidental modification and improve standard uses through the team or company, these libraries are stored separately.

© Copyright 2019 Xilinx

2-2-7. Click the **xil_defaultlib** entry in the Library field of the *time_utilities_pkg.vhd* file and enter **utilities_lib**.

2-2-8. Place both *string_utilities* packages into *utilities_lib* by selecting the proper library from the Library drop-down list.

2-2-9. Click the **xil_defaultlib** entry in the Library field of the *fifo_pkg.vhd* file and enter **specific_support_lib**.

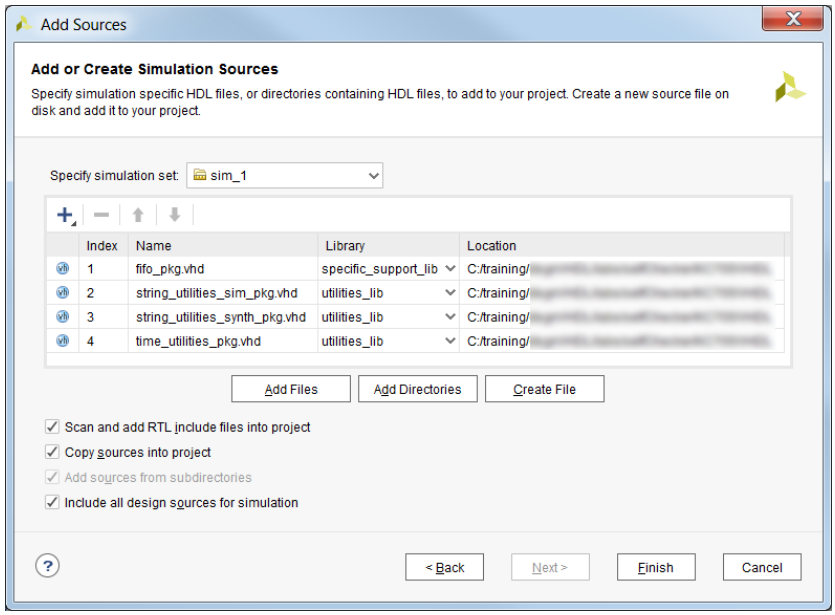


Figure 10-2: Associating the Library

2-2-10. Click **Finish** to complete the loading of the packages into the project.

Question 1

Describe the structure of the project now that the libraries have been created. Identify where the packages are located in the file system.

© Copyright 2019 Xilinx

Coding the Stimulus Step 3

A testbench just for the UART_rx portion of the design (which uses all the modules required for the UART_led testbench) was written for another project. As is commonly done, similar modules can be modified to suit specific purposes. This is one form of code reuse. A testbench for testing just the UART_rx module has been provided for you. Your job is to modify this testbench to work with the UART_led design.

This implies several modifications:

- The name of the module and entity will need to be changed to reflect the new purpose of this module.
- UART_rx is no longer the UUT—the UART_led design is.
- The response checker expects a one clock cycle-wide pulse indicating that there is new data present. UART_rx provides this signal while UART_led does not.

You will now add the tb_uart_driver.vhd and tb_resp_checker.vhd files to the project.

3-1. Add simulation files to the design.

3-1-1. Click **Add Sources** under Project Manager in the Flow Navigator.

The Add Sources Wizard opens.

3-1-2. Select **Add or create simulation sources**.

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

3-1-4. Click the **Plus** (+) icon and select **Add Files**.

3-1-5. Browse to the `$selfChecker/lab/KCU105` directory if it is not open already.

3-1-6. Select **tb_uart_driver.vhd and tb_resp_checker.vhd**.

3-1-7. Click **OK**.

3-1-8. Click **Finish** to add the file(s) to the project.

© Copyright 2019 Xilinx

3-2. Change the name of the existing testbench from test_uart_rx.vhd to UART_led_full_tb. Make the appropriate changes to the entity and architecture as well. Add to the project.

3-2-1. Click **Add Sources** under Project Manager in the Flow Navigator.

The Add Sources Wizard opens.

3-2-2. Select **Add or create simulation sources**.

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

3-2-4. Click the **Plus** (+) icon and select **Add Files**.

3-2-5. Click on the **test_uart_rx.vhd** file and rename it to **UART_led_full_tb.vhd**.

3-2-6. Select the renamed file **UART_led_full_tb.vhd** and click **OK**.

3-2-7. Click **Finish** in the Add or Create Simulation Sources dialog box to add the file to the project.

3-2-8. Select the **Sources** view.

3-2-9. Expand **Simulation Sources** and double-click the **UART_led_full_tb.vhd** to open it in a text editor.

3-2-10. Replace each occurrence of *test_uart_rx* with **UART_led_full_tb**.

This can be done using the global search and replace function or manually as there are only three occurrences.

3-3. Change the UUT from the uart_rx module to the uart_led module.

Although suggested code solutions can be found in the Answers section, you should make every effort to code these yourself.

3-3-1. Comment the instantiation and declaration of uart_rx by clicking the **Comment** (//) icon in the toolbar of the text editor.

```
      -- Instantiate the unit under test
--        uart_rx_i0: uart_rx
--          generic map (
--                  BAUD_RATE   => BAUD_RATE,
--                  CLOCK_RATE  => CLOCK_RATE                -- freq of clk
--              )
--           Port map ( rst_clk_rx      => rst_clk_rx,
--                      clk_rx          => clk_rx,
--                      rxd_i           => rxd_clk_rx,
--                      rx_data         => rx_data,
--                      rx_data_rdy     => rx_data_rdy,
--                      frm_err         => frm_err
--                    );
```

Figure 10-3: Removing the Old UUT Instantiation

Similarly remove the component declaration of uart_rx.

3-3-2. Insert the new UUT instantiation; i.e., name the UUT as UART_led (from UART_rx to UART_led).

3-3-3. Add the uart_led component and generics manually.

3-3-4. Connect the generics and ports as necessary.

© Copyright 2019 Xilinx

Now that there is an instantiation, it must be declared. Often this is done first; however, the order is not relevant as long as it is done prior to analysis.

3-3-5. Write the code to create the declaration for the UUT.

Note: It is not compulsory to comment out or delete the old UART_rx declaration, as declarations do not create logic. Once changes are made and the simulation runs successfully, you can revisit the testbench and clean it up by removing blocks of unused code.

The signal sel_pin should be connected to gnd. Because gnd(0) was used in the instantiation, define gnd(0) in the architecture declaration section.

Note: You can find the answer in the Answer section.

3-4. Write the code that provides the response checker module with a single clock period pulse each time that the output from the UUT changes.

Note: **This will fail if the same two characters are transmitted back to back.**

Although suggested code solutions can be found in the Answers section, you should make every effort to code these yourself.

3-4-1. Add a process *or* combinatorial statement(s) in the Architecture description section of the code to provide a one-clock wide pulse to the response checker module.

The combinatorial solution requires that both last_datum (std_logic_vector(7 downto 0) and new_data_present (std_logic) need to be defined.

The process solution requires that only new_data_present (std_logic) needs to be defined.

3-4-2. Make the appropriate signal definitions in the Architecture declaration area.

Make sure that the rx_data_rdy signal of tb_resp_checker is connected to new_data_present.

Also make sure that the frm_err signal of tb_resp_checker is connected to gnd(0).

Question 2

If both signals are defined and the process solution is used, what is the effect of having last_datum defined as a signal?

_____

_____

3-4-3. Select **File** > **Text Editor** > **Save File**.

© Copyright 2019 Xilinx

Simulating the Design Step 4

In the final portion of this lab, you will run the simulation and read the simulation output.

4-1. Select uart_led_full_tb and launch the Vivado simulator.

4-1-1. Select **Settings** in the Flow Navigator.

Select **Simulation** in the Settings dialog box.

4-1-2. Select the **Simulation** tab.

4-1-3. Enter **6ms** in the Simulation Run Time field.

4-1-4. Click the **xsim.simulate.wdb** field and enter `/home/xilinx/training/selfChecker/lab/KCU105/selfChecker.wcfg`.

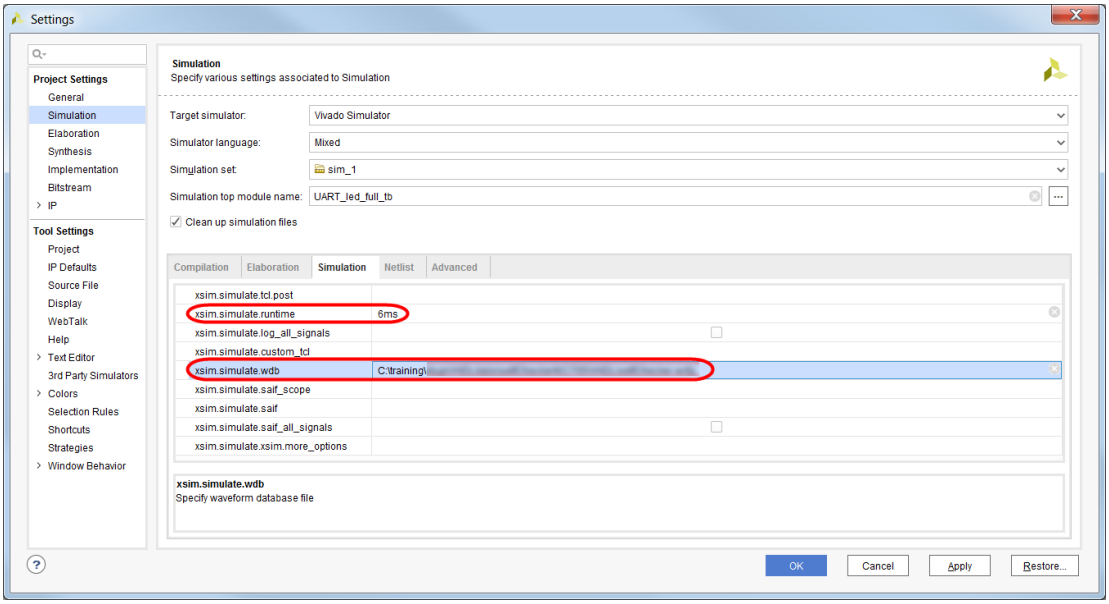


Figure 10-4: Setting New Default Simulation Run Time

4-1-5. Click **OK**.

4-1-6. Select **Simulation** > **Run Simulation** > **Run Behavioral Simulation** in the Flow Navigator to run the simulation.

Note: The simulation should self-terminate prior to 6 ms.

Question 3

How is the output of the simulation viewed?

_____

_____

© Copyright 2019 Xilinx

Question 4

How is the simulation ended? Locate the code and explain its behavior.

_____

_____

Question 5

What are the relevant messages? What do they mean?

_____

_____

Question 6

What role do the packages play? That is, how do they interact with the testbench?

_____

_____

4-2. Close the simulation.

4-2-1. Select **Simulation** in the Flow Navigator, then right-click and select **Close Simulation**.

4-2-2. Click **OK** to close the simulation.

4-3. Close the Vivado Design Suite.

4-3-1. Select **File** > **Exit**.

The Exit Vivado dialog box opens.

Figure 10-5: Exit Vivado Dialog Box

4-3-2. Click **OK**.

© Copyright 2019 Xilinx

[Optional]: Clean up the file system.

Some systems 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. Both processes will recursively delete all of the files in the `$selfChecker` directory.

4-4. Clean up the VirtualBox file system via the GUI.

4-4-1. Using the graphical browser (Windows: press the <**Windows**> key + <**E**>; Linux: press <**Ctrl** + **N**>), navigate to `$selfChecker`.

4-4-2. Select **`selfChecker`**.

4-4-3. Press <**Delete**>.

4-5. Clean up the VirtualBox file system via the command line.

4-5-1. Open a terminal window (Windows: press the <**Windows**> key + <**R**>, then enter **cmd**; Linux: press <**Ctrl** + **Alt** + **T**>)

4-5-2. Enter the following command to delete the contents of the workspace:

[Windows users]: **`rd /s /q $selfChecker`**

[Linux users]: **`rm -rf $selfChecker`**

Summary

This lab walked you through a typical reuse scenario—a testbench that was used for a prior, but similar, project was modified for use with the current UART_led design.

The testbench, like the simpler version in the "Simulating a Simple Design" lab, generates data and serializes it for the UART_rx portion of the design.

However, this time, a FIFO module was created to provide time independence and it was attached to a response checker that compared a delayed version of the transmitted data with the output of the UUT. The checker indicated if there was a match, or if the character was received incorrectly.

© Copyright 2019 Xilinx

Answers

1. Describe the structure of the project now that the libraries have been created. Identify where the packages are located in the file system.

   The files are located in the top of the project directory. The "libraries" are virtual constructs within the Vivado software. Many designers will create separate subdirectories for each package, place the packages into the appropriate subdirectory, then add the packages to the virtual library structure as you just did in the previous step.

2. If both signals are defined and the process solution is used, what is the effect of having last_datum defined as a signal?

   Absolutely none: Last_datum as a signal goes unused because the variable last_datum in the process is used and scoping rules treat these as two completely different objects.

3. How is the output of the simulation viewed?

   As always, the waveform view is available. However, the power in this type of testbench comes from the automated messages that are displayed in the console window.

4. How is the simulation ended? Locate the code and explain its behavior.

   Simulation is ended in the tb_uart_driver module 1 ms after the last bit is transmitted. Although not an ideal situation as it assumes that the latency through the UUT is less than 1 ms, it is satisfactory for this type of testbench.

   A better way would be to send a specific character, which indicates the end of data, and have the response checker halt simulation when this final character is validated.

   The final message is reported as a failure error, which it is NOT. The failure error severity causes the simulator to halt. The report portion of that assertion message indicates that it is not actually a failure, but rather all the data has been transmitted and received.

5. What are the relevant messages? What do they mean?

   The messages are time stamped as to exactly when the notated event occurred. This is done so that if there is a discrepancy, you can "jump" to the by restarting the simulation and running for just under the time listed in the tag. From there, the code can be single-stepped to locate the source of the problem.

   Both the character, as it is generated, and the character, when it is received, are listed and displayed in ASCII and hexadecimal.

© Copyright 2019 Xilinx

6. What role do the packages play? That is, how do they interact with the testbench?

   The packages provide a convent extension to the capabilities of VHDL.

   string_utilities_sim_pkg leverages some technique used in the Advanced VHDL class to provide 'C' language-like function calls. Native VHDL contains some less than impressive string and handling functions. This package, although only valid within the simulation environment, provides a small set of functions to ease string manipulation in VHDL.

   string_utilities_synth_pkg is a synthesizable collection of functions and constants that provide higher level conversions among integers, std_logic_vectors, and characters.

   time_utilities_pkg offers a small assortment of functions that make console writing a bit easier, including "writeNowToScreen" which time stamps messages using engineering notation. This module is not synthesizable.

   fifo_pkg implements a simple, non-synthesizable FIFO mechanism enabling you to use simple calls to push and pop rather than have to worry about complex signal timing to implement the same function.

   These packages are provided as a courtesy from Xilinx and are not fully evaluated. You are welcome to use them as you see fit and at your own risk.

© Copyright 2019 Xilinx

Changing the UUT (from UART_rx to UART_led)

New UUT instantiation:

```
 -- Instantiate the unit under test
UUT: uart_led
   PORT MAP(
      clk_pin  => ,
      rst_pin  => ,
      rxd_pin  => ,
      sel_pin  => ,
      led_pins =>
   );
```

Adding the generics:

```
 -- Instantiate the unit under test
UUT: uart_led
   Generic map (
       CLOCK_RATE => ,
       BAUD_RATE  =>
      )
   PORT MAP(
      clk_pin  => ,
      rst_pin  => ,
      rxd_pin  => ,
      sel_pin  => ,
      led_pins =>
   );
```

Connecting the generics and ports:

```
 -- Instantiate the unit under test
UUT: uart_led
   Generic map (                    -- values from test-bench's generics
       CLOCK_RATE => CLOCK_RATE,
       BAUD_RATE  => BAUD_RATE
      )
   PORT MAP(
      clk_pin  => clk_rx,
      rst_pin  => rst_clk_rx,
      rxd_pin  => rxd_clk_rx,
      sel_pin  => gnd(0),
      led_pins => rx_data
   );
```

Declaration for the UUT:

```
  -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT uart_led
    Generic (CLOCK_RATE : integer := 50_000_000;
             BAUD_RATE  : integer :=    115_200
            );
    PORT(
       clk_pin : IN std_logic;
       rst_pin : IN std_logic;
       rxd_pin : IN std_logic;
       sel_pin : IN std_logic;
       led_pins : OUT std_logic_vector(7 downto 0)
    );
 END COMPONENT uart_led;
```

Defining gnd(0):

```
signal   gnd : std_logic_vector(7 downto 0) := (others=>'0');
```


© Copyright 2019 Xilinx

Coding the Response Checker Module

Process *or* combinatorial statement(s) to provide a one-clock wide pulse to the response checker module.

Combinatorial solution:

```
-- combinatorial solution to provide the response checker with a one clock period pulse to indicate new data
new_data_present <= '1' when (last_datum /= rx_data) else '0'; -- when data changes then assert new_data_present
last_datum       <= rx_data when rising_edge(clk_rx);          -- register to create a one clock delay
```

Process (sequential) solution:

```
-- sequential solution to provide the response checker with a one clock period pulse to indicate new data
mkDataAvail: process (clk_rx)
      variable last_datum  : std_logic_vector(7 downto 0) := (others=>'U');
   begin
      if rising_edge(clk_rx) then
         if (rx_data /= last_datum) then
            new_data_present <= '1';
         else
            new_data_present <= '0';
         end if;
         last_datum := rx_data;
      end if;
   end process mkDataAvail;
```

Signal definitions:

```
-- code for pulse generation when new data is available
signal new_data_present   : std_logic := 'U';
signal last_datum         : std_logic_vector(7 downto 0) := (others=>'U');
```

Note: This code is also valid for the process solution.

© Copyright 2019 Xilinx

Lab 11: Xilinx Tool Flow

2019.1

Abstract

This lab introduces you to the Vivado® Design Suite tool flow that you will use during the development of your design.

This lab should take approximately 60 minutes.

Objectives

After completing this lab, you will be able to:

- Create a new project in the Vivado Design Suite
- Add design files to a project
- Use the Vivado simulator to simulate a design
- Generate a bitstream
- Use the Vivado hardware manager to download a bitstream to a board

Introduction

This Xilinx tool flow lab will add the VHDL source code that has been provided to you, create the simulation environment, and introduce you to the following Vivado features:

- Vivado synthesis
- Vivado implementation
- Bitstream generation

The Vivado synthesis and implementation features are timing driven and use XDC format constraints. Various reports and analysis features are available at each stage of the design process. You can run the design through the entire flow by using the Vivado IDE, by using batch Tcl scripts, or by entering Tcl commands at the Vivado Design Suite Tcl shell or at the Vivado IDE Tcl Console.

After implementation, the device can be programmed and then analyzed within the Vivado Design Suite environment. Debug signals can be easily identified in RTL or after synthesis and are processed throughout the flow. Debug cores can be configured and inserted either in RTL or in the synthesized netlist. The Vivado logic analyzer also enables hardware validation. The interface is designed to be consistent with the Vivado simulator, and both share a common waveform viewer.

© Copyright 2019 Xilinx

The Vivado IP integrator is the main interface for creating embedded designs using Zynq® All Programmable SoC devices or MicroBlaze™ processors.

In implementation, the `opt_design`, `place_design`, and `route_design` processes will be performed. The `place_design` process finds the physical locations for each and every resource in the final netlist. Guided by the user's placement and timing constraints, this will try many configurations to meet the minimum performance specifications.

If the place and route algorithm cannot find a solution, a warning will be generated and you will have to open the Timing Report to identify which paths failed to meet timing and determine an appropriate solution.

The last step in the flow is to convert the output from the place and route tools into a bitstream and program (or download) this bitstream to the device.

Understanding the Lab Environment

The labs and demos provided in this course are designed to run under the provided Linux platform. The provided Linux platform has a customized lab environment. These customizations have been made to simplify and enhance your learning experience. Many labs and demos can be successfully executed in the Windows environment as well. Topics including PetaLinux, Yocto, and QEMU are generally not supported under Windows.

The instructions, for the sake of brevity, are expressed using the Linux notation. This includes the forward slash ('/') as the hierarchy separator instead of the Windows backslash ('\'). For students who want to run under Windows, it is their responsibility to use the correct hierarchy separator.

Windows users can run Linux using the VirtualBox tool from Oracle. Details on how to acquire and install this tool can be found in the lab setup guide. The Xilinx Customer Education group offers a preconfigured virtual machine image and includes all the tools necessary to run the labs. Some tools and IP do require licenses, which are the responsibility of the student.

Several environment variables, described in detail in the lab setup guide, which also discusses how you can customize the environment for your situation, are provided for ease of use. Most notably, many environment variables have been defined to reduce the amount of typing you need to do when entering paths.

These environment variables are:

- XILINX_PATH - points to the installation directory for the Xilinx tools. Tools such as the Vivado Design Suite and SDK can be found here.
- PETALINUX_PATH - points to the installation directory for the Xilinx PetaLinux tools. Often this is within XILINX_PATH, but some choose to install it elsewhere.
- TRAINING_PATH - points to the space allocated for students to work through their labs. This directory includes prebuilt images and starting points for the labs and demos.
- flowLab - points to the specific name of the topic within the TRAINING_PATH directory.

  This lab uses $flowLab to indicate the directory for this lab, which is equivalent to $TRAINING_PATH/flowLab.

© Copyright 2019 Xilinx

These environment variables work well in the Linux terminal and the Vivado Design Suite and scripts, but not in Eclipse-based tools such as the SDK tool and the SDx™ environment. When using the Vivado tools, you can use $flowLab and the tools will automatically expand the environment variable for you.

However, when you see the notation $flowLab/workspace in SDK or the SDx environment, you should interpret it in its expanded form. The expanded form can be found by opening a terminal window and enter "echo $flowLab".

This lab assumes that you are starting with a Linux environment. If you are working on a Windows system, refer to the lab setup guide to see how to use the Oracle VirtualBox application to run the Xilinx Customer Education pre-configured Ubuntu Linux virtual machine.

General Flow

Creating a New Project Step 1

1-1. Launch the Vivado Design Suite.

If you do not recall how to perform this task, refer to the "Launching the Vivado Design Suite" section under Vivado Design Suite Operations in the *Lab Reference Guide*.

1-2. Create a new Vivado Design Suite project named *flowLab* and locate it in the following directory.

Browse to the `$flowLab/lab/KCU105` directory.

Target the project for the Kintex UltraScale KCU105 Evaluation Platform evaluation board.

If you do not recall how to perform this task, refer to the "Creating a Blank Vivado Design Suite Project" section in the *Lab Reference Guide*.

From the settings in the Flow Navigator, change the target language to VHDL.

© Copyright 2019 Xilinx

Adding New Design Files to the Project Step 2

You will now add design source files from the working directory into the project.

The Vivado IDE allows different file types to be added as design sources, including Verilog, VHDL, SystemVerilog, EDIF, and XDC constraint files. These are displayed categorically in the Sources window. The tabs at the bottom of the Sources window enable the files to be displayed by hierarchy, libraries, or compile order.

Add the following design sources files to the project:

- *led_ctl.vhd*
- *meta_harden.vhd*
- *uart_baud_gen.vhd*
- *uart_led.vhd*
- *uart_rx.vhd*
- *uart_rx_ctl.vhd*

HDL source files can be added to the design at any time.

2-1. Add an HDL source file to the design.

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

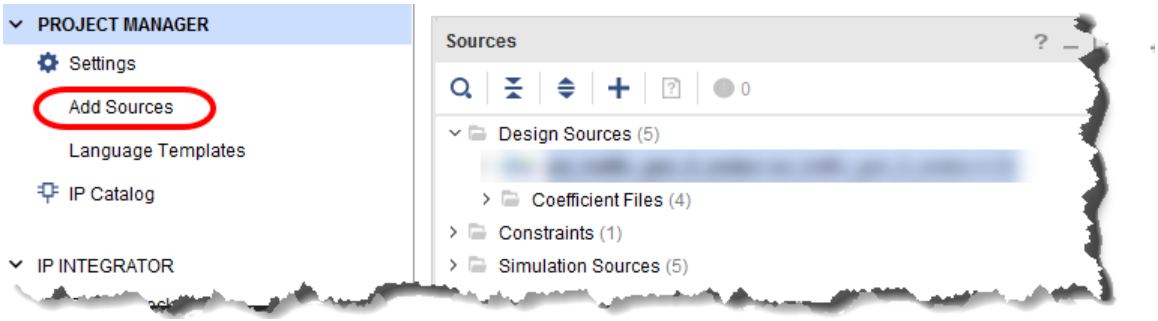


Figure 11-1: Selecting Add Sources

The Add Sources dialog box opens, allowing you to add HDL source files to the project.

© Copyright 2019 Xilinx

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

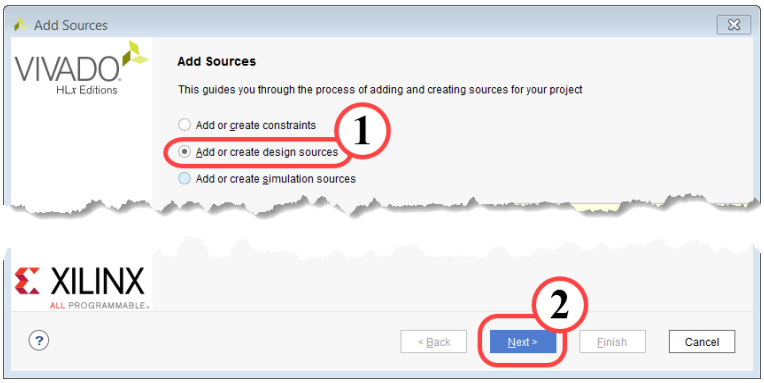


Figure 11-2: Selecting Add or Create Design Sources

2-1-3. Click **Next** to begin selecting source files (2).

The Add or Create Design Sources dialog box opens and prompts you to add existing HDL source files or to create new HDL sources files.

2-1-4. Click the **Plus** (+) icon and select **Add Files**.

2-1-5. Browse to `$flowLab/lab/KCU105` if it is not open already.

2-1-6. Select **the files listed above**.

2-1-7. Double-click the source file name in the Add Source Files dialog box to select the file(s) or click **OK**.

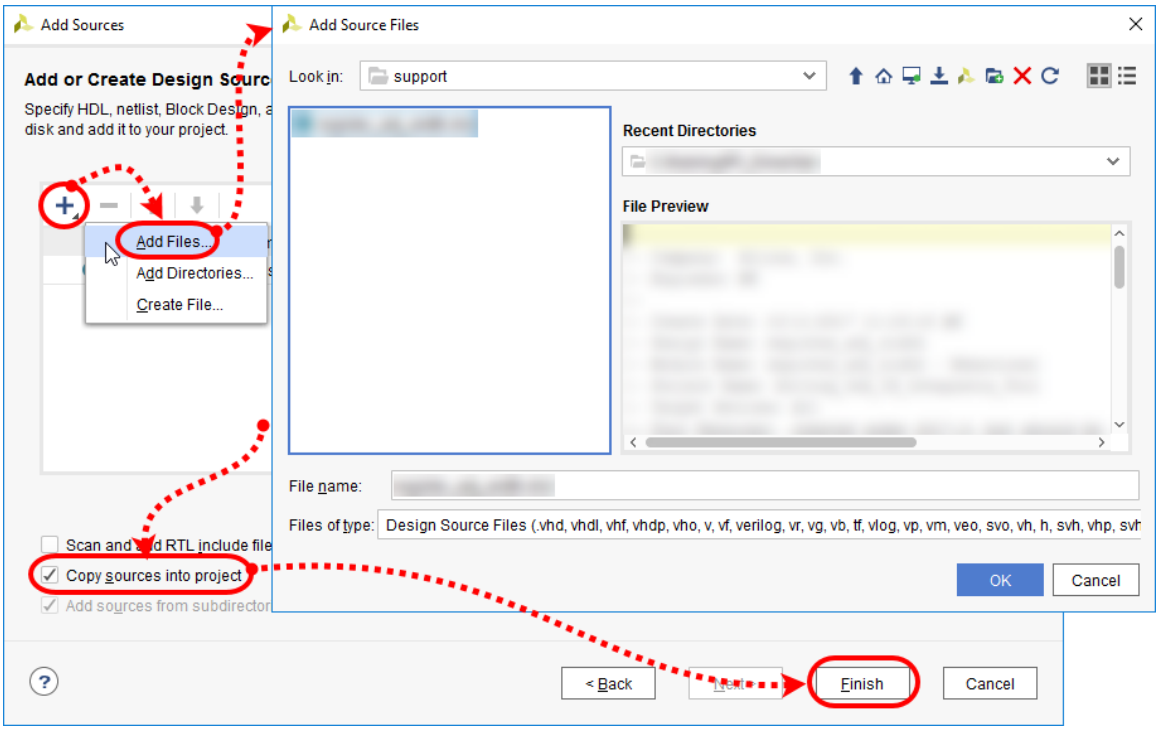


Figure 11-3: Selecting Add Files

© Copyright 2019 Xilinx

2-1-8. Ensure that the **Copy sources into project** option is selected (when building IP this will be listed as **Copy sources into IP Directory**).

2-1-9. Click **Finish** in the Add or Create Design Sources dialog box to add the HDL sources to the project.

2-2. Add the testbench files from the working directory into the project.

2-2-1. Click **Add Sources** in the Flow Navigator.

The Add Sources Wizard opens.

2-2-2. Select **Add or create simulation sources** to add the simulation source files to the project.

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

The Add or Create Simulation Sources dialog box opens.

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

2-2-5. Select the following files to add to the project:

- *tb_resp_checker.vhd*
- *tb_uart_driver.vhd*
- *tb_fifo_pkg.vhd*
- *test_uart_rx.vhd*
- *string_utilities_sim_pkg.vhd*
- *string_utilities_synth_pkg.vhd*
- *time_utilities_pkg.vhd*

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

© Copyright 2019 Xilinx

2-3. Associate the VHDL package files to their corresponding libraries.

2-3-1. Click next to the *string_utilities_sim_pkg.vhd* file in the Library tab and enter **utilities_lib**. Do the same for the *string_utilities_synth_pkg.vhd* and *time_utilities_pkg.vhd* files

2-3-2. Enter **specific_support_lib** in the Library tab next to the *tb_fifo_pkg.vhd* file.

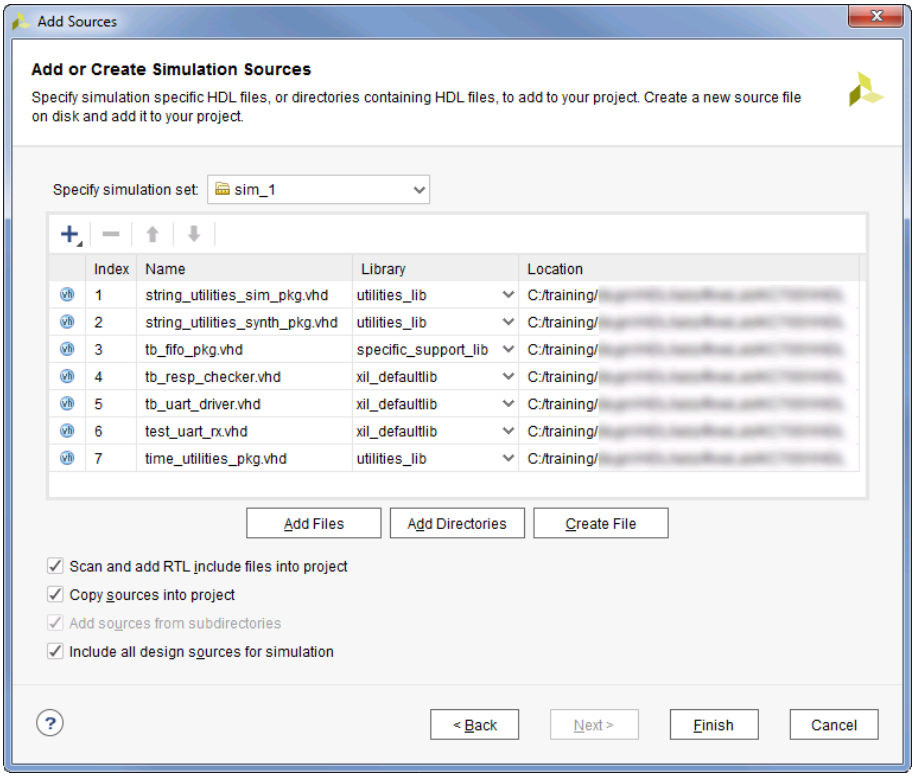


Figure 11-4: Add or Create Simulation Sources Dialog Box with Files and Libraries Associated

2-3-3. Click **Finish**.

© Copyright 2019 Xilinx

2-3-4. Select the **Libraries** tab in the Sources window to verify that the libraries have been assigned properly.

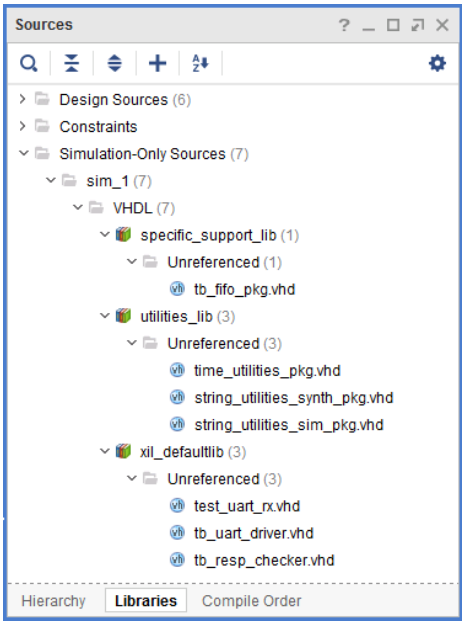


Figure 11-5: Viewing Properly Assigned Libraries

The following package files should be visible in the utilities_lib library:

- *string_utilites_synth_pkg.vhd*
- *string_utilites_sim_pkg.vhd*
- *time_utilites_pkg.vhd*

The following package file should be visible in the specific_support_lib library.

- *tb_fifo_pkg.vhd*

2-4. Add the constraints file (XDC) from the working directory into the project.

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

The Add Sources Wizard opens.

2-4-2. Select **Add or create constraints** to add the simulation source files to the project.

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

The Add or Create Constraints dialog box opens.

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

2-4-5. Browse to the project working directory.

This should be the default directory when you click Add Files.

2-4-6. Add the **uart_led.xdc** file to the project and click **Finish**.

© Copyright 2019 Xilinx

2-5. Analyze the source files and constraints used in the project.

2-5-1. Select the **Hierarchy** tab in the Sources window and click the **Expand All** (⇕) icon to view all the files.

To easily view the contents of any window, click the **Maximize** icon.

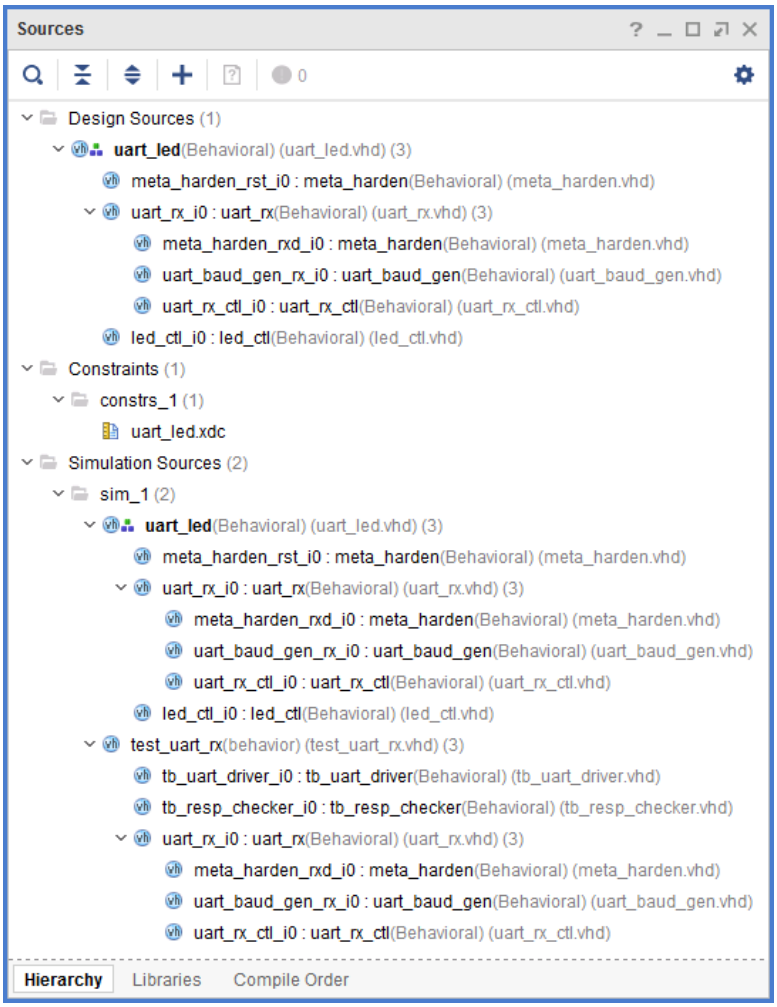


Figure 11-6: Viewing Sources

The *Design Sources* folder helps you keep track of VHDL and Verilog design source files and libraries.

The *Constraints* folder helps you keep track of the constraints files.

The *Simulation Sources* folder helps you keep track of VHDL and Verilog simulation source files and libraries. Notice that the design hierarchy is displayed by default.

In the Libraries tab, sources are grouped by file type, while the Compile Order tab shows the file order used for synthesis.

© Copyright 2019 Xilinx

Simulating the Design Step 3

You will now perform behavioral simulation. The Hierarchy window shows you all top-level modules (and the trees below them), some of which will be test fixtures, others are un-instantiated sub-modules.

Open the *test_uart_rx.vhd* stimulus file. You will see that a number of ASCII characters are being sent serially to the *uart_rx* module, which de-serializes them.

Start the simulation. After it starts, delete all the existing waves on the waveform viewer, and add the following signals:

- *test_uart_rx/data_to_send [7:0]:* Format ASCII
- *test_uart_rx/uart_rx_io/clk_rx:* Format binary
- *test_uart_rx/uart_rx_io/rx_data [7:0]:* Format ASCII
- *test_uart_rx/uart_rx_io/rx_data_rdy:* Format binary
- *test_uart_rx/uart_rx_io/rxd_i:* Format binary
- *test_uart_rx/uart_rx_io/frm_err:* Format binary
- *test_uart_rx/tb_resp_checker_i0/char_received [7:0]:* Format ASCII

The signal *data_to_send* is the character to be sent by the testbench. The signal *rxd_i* is the RS232 serial signal generated by the testbench and is the input to the *uart_rx* module. The signals *rx_data*, *rx_data_rdy*, and *frm_err* are the outputs of the *uart_rx* module. The signal *data_received* is the data received by the testbench to verify.

3-1. Explore the *test_uart_rx.vhd* stimulus file.

3-1-1. Right-click **test_uart_rx - behavior (test_uart_rx.vhd)** in **Simulation Sources** > **sim_1** and select **Set as Top**.

3-1-2. Double-click **test_uart_rx.vhd**.

Review this simple testbench. Notice how the clock is being generated and how the transmission of serial characters is achieved. If you have any questions, ask the instructor.

© Copyright 2019 Xilinx

3-2. Simulate the design.

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

3-2-2. Select **Run Behavioral Simulation**.

The tool will compile the testbench and launch the Vivado simulator.

Note that the Vivado IDE view changes and is now populated with the following views required for simulation.

Scopes

This window has multiple tabs. The Scopes tab shows all the instances in the current simulation.

Objects

This window shows the objects that exist within the selected instance. These objects can be dragged into the waveform viewer.

Main Window

By default, an Untitled Waveform viewer will appear displaying only the signals at the top level of the testbench.

3-2-3. Remove the existing signals in the waveform viewer by using Shift-select and pressing **<Delete>** (or right-click and select **Delete**).

3-2-4. Select **test_uart_rx** in the Scope view of the Vivado IDE GUI.

3-2-5. Right-click **data_to_send[7:0]** in the Objects view and select **Add To Wave Window**.

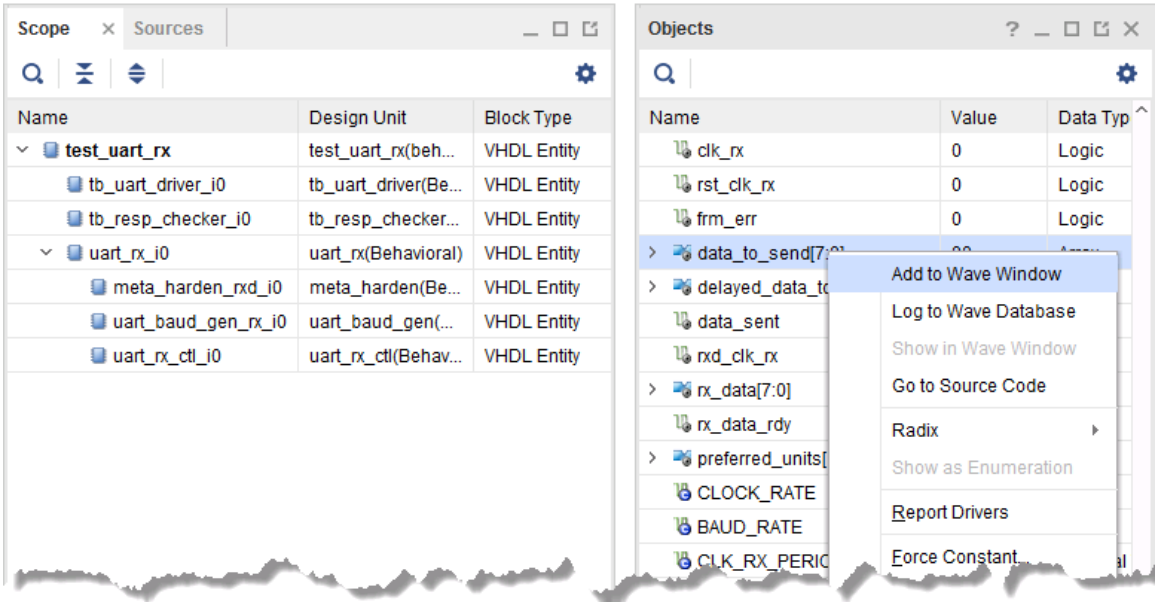


Figure 11-7: Selecting test_uart_rx to Drag data_to_send

© Copyright 2019 Xilinx

3-2-6. Right-click **data_to_send[7:0]** value column in the Waveform window and select **Radix** > **ASCII**.

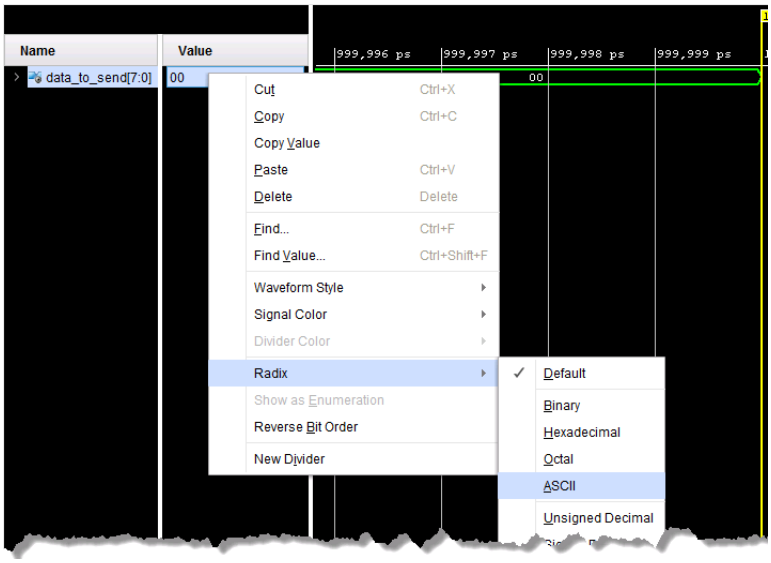


Figure 11-8: Changing the Radix of char_to_send

3-2-7. Select **test_uart_rx** > **uart_rx_i0** in the Scopes view. Drag **clk_rx**, **rxd_i**, **rx_data[7:0]**, **rx_data_rdy**, and **frm_err** to the Waveform viewer. Change the radix of **rx_data[7:0]** to ASCII and leave the others at their default.

3-2-8. Select **tb_resp_checker_i0** and drag **char_received[7:0]** to the Waveform viewer. Change its radix to ASCII as well.

3-3. Restart and run the simulation to completion.

3-3-1. Click the **Restart** ( ) icon.

3-3-2. Click the **Run All** ( ) icon in the menu bar.

3-3-3. Confirm the following:

- The Console window shows that all characters were sent and received and that no errors occurred. Errors contain the word "ERROR" right after the simulation time.
- The string to be sent is appearing on **data_to_send**.
- The serial data appears on **rxd_i**.
- The characters are received correctly by **uart_rx** and are visible and stable on **rx_data** when **rx_data_rdy**.
- The string received is appearing on **char_received**.

You will need to use the slider at the right of the Console window to see earlier messages and the slider at the bottom of the Waveform window and the Zoom Control icons in the menu bar to zoom and pan the wave display.

3-3-4. Select **File** > **Close Simulation** after confirming the above.

© Copyright 2019 Xilinx

3-3-5. Click **OK** in the **Confirm Close** window.

3-3-6. Click **Discard** in the pop-up **Save Waveform Configuration** dialog box.

Question 1

Why was running a simple simulation included in this lab?

Implementing and Analyzing the Implemented Design Step 4

Here you will implement the design and generate the utilization report to observe the design utilization.

4-1. Implement the design.

4-1-1. Click **Run Implementation** in the Flow Navigator (under Implementation).

Alternatively, you can select **Flow** > **Run Implementation**.

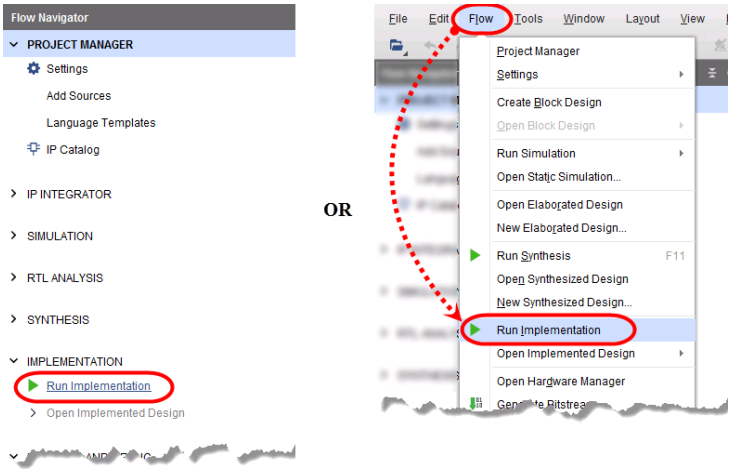


Figure 11-9: Selecting Run Implementation

Note: If needed, click **OK** to launch synthesis first if prompted.

Notice that the tools run all of the processes required to implement the design. This means that if the synthesized netlist is not available already, the Vivado Design Suite will run synthesis before running implementation.

After implementation completes, the Implementation Completed dialog box opens. The dialog box prompts you to open the implemented design, generate the bitstream, or view reports.

© Copyright 2019 Xilinx

4-1-2. Click **OK** again to launch the selected runs.

4-1-3. Select **Open Implemented Design**.

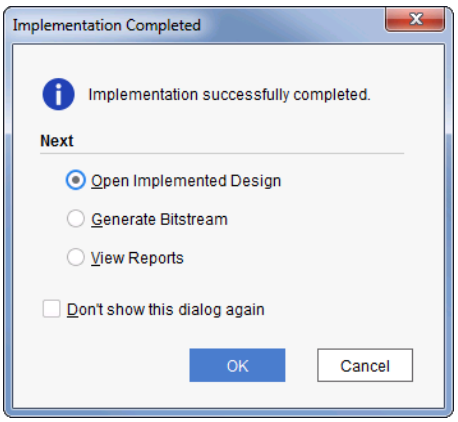


Figure 11-10: Implementation Completed Dialog Box

4-1-4. Click **OK**.

4-2. Generate a Utilization report on the implemented design.

4-2-1. Click **Report Utilization** under Implemented Design in the Flow Navigator.

Alternatively, you can select **Reports** > **Report Utilization**.

The Report Utilization dialog box opens.

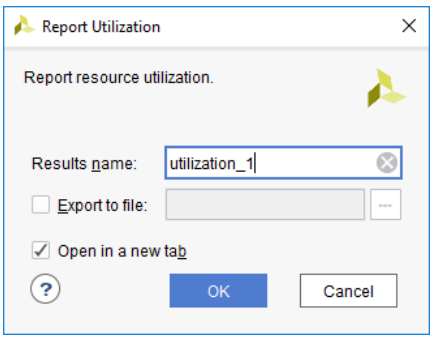


Figure 11-11: Report Utilization Dialog Box

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

The Design Utilization Summary window opens at the bottom in the Utilization tab.

Note that the FPGA resources are listed in a hierarchical format for each RTL design module.

The Utilization report is displayed at the bottom of the Vivado IDE. You can select any of the resources on the left to view its corresponding utilization.

4-2-3. Explore the Utilization report.

4-2-4. Close the Utilization report.

© Copyright 2019 Xilinx

Downloading the Bitstream Step 5

You will now generate the bitstream and download it onto the KCU105 board.

5-1. Generate the bitstream.

5-1-1. Click **Generate Bitstream** in the Flow Navigator under Program and Debug.

Alternatively, you can select **Flow** > **Generate Bitstream**.

5-2. Power on the KCU105 board.

5-2-1. Connect a USB cable from a USB port on your computer to the USB UART connector on the KCU105 evaluation board.

5-2-2. Connect the USB cable to the Digilent USB JTAG interface.

5-2-3. Ensure that the power cord is plugged in and turn on the evaluation board.

5-2-4. Make sure that the board settings are proper.

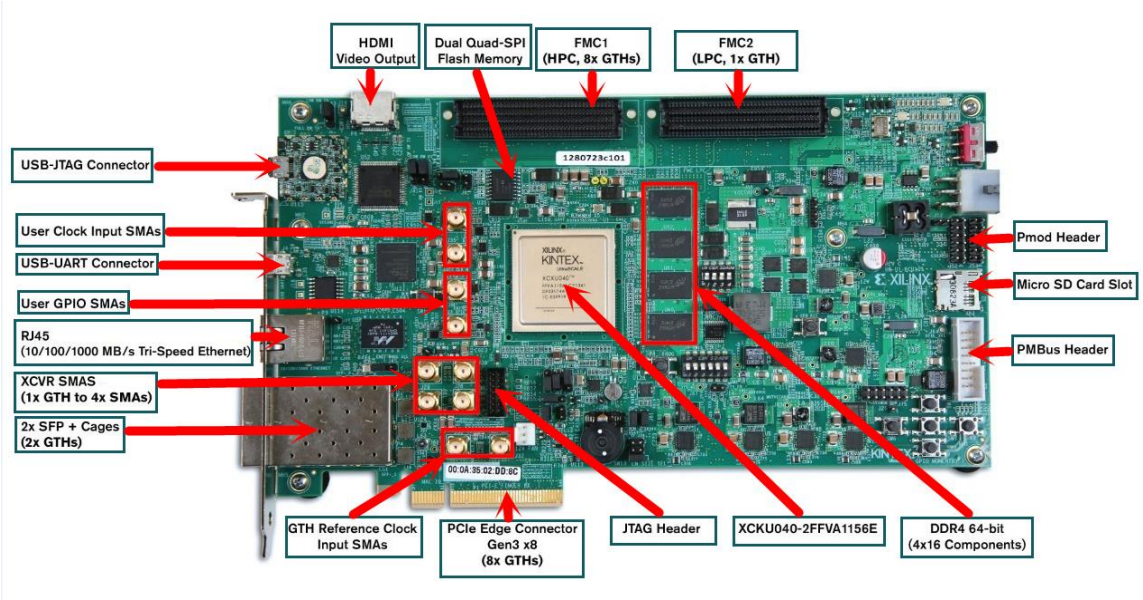


Figure 11-12: KCU105 Evaluation Board

You may be prompted to install drivers when the board is first connected. Do not allow the driver installation to search the Web, but allow it to search for the drivers on your computer.

5-2-5. Ensure that all DIP switches (SW11) are in the off position.

© Copyright 2019 Xilinx

A hardware manager is the portion of the Vivado Design Suite that enables the monitoring of cores that were added to a design.

5-3. Open the hardware manager.

5-3-1. Click **Open Hardware Manager** in the Bitstream Completed dialog box and click **OK**.

The Hardware Manager window opens.

The hardware needs to be connected and the information bar invites you to open an existing or a new target.

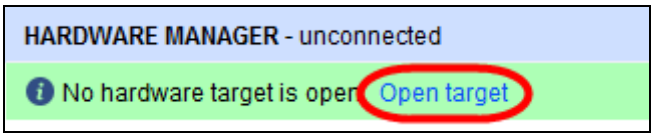


Figure 11-13: Opening a Hardware Target

5-4. Connect the target through the New Target Wizard to guide you through the process.

5-4-1. Click **Open target** > **Open New Target**.

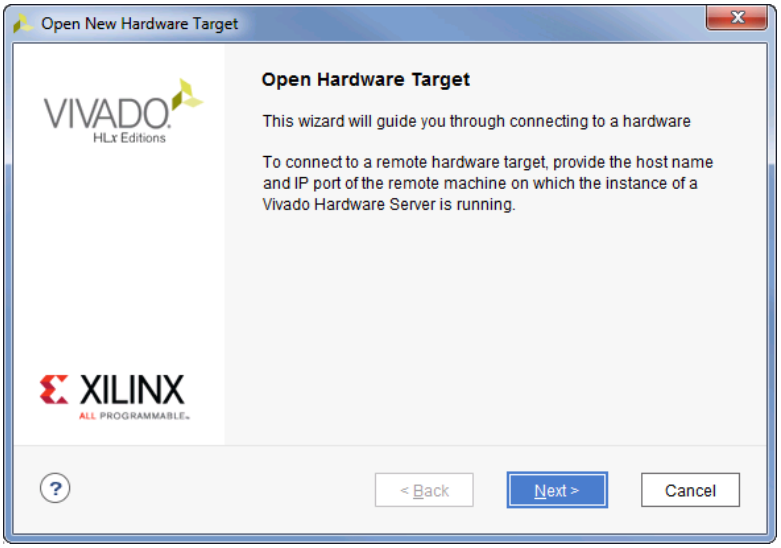


Figure 11-14: Open Hardware Target Dialog Box

5-4-2. Click **Next** to set the hardware server settings.

5-4-3. Enter a name for the server.

© Copyright 2019 Xilinx

Typically this is left at its default value.

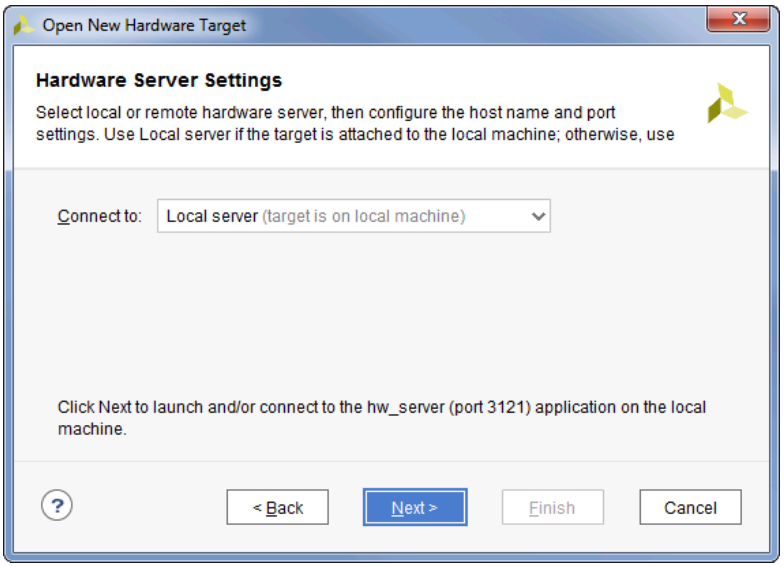


Figure 11-15: Setting the Server Name for the New Hardware Target

5-4-4. Click **Next** to select the hardware target.

5-4-5. Verify the hardware target.

This becomes important when there are multiple targets connected to the PC.

You can change the frequency of the JTAG cable if you are experiencing communications problems.

[Linux users]: If you receive an error saying that no active target is found, check the USB connections by selecting **Devices** > **USB** in the VirtualBox toolbar at the top.

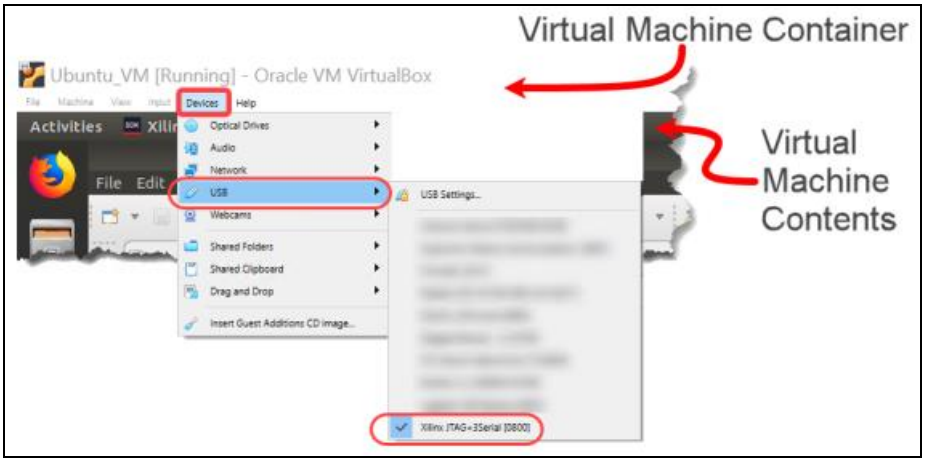


Figure 11-16: Selecting the Hardware Target

© Copyright 2019 Xilinx

5-4-6. Leave the frequency at its default value.

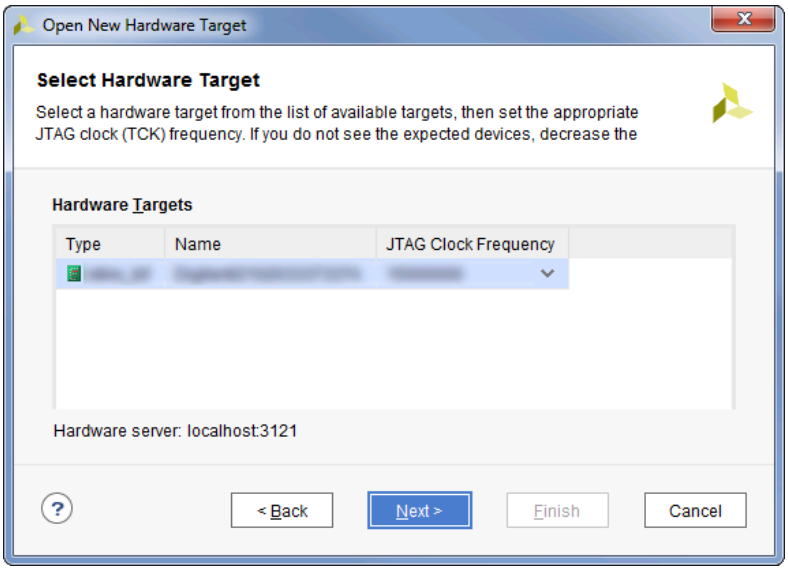


Figure 11-17: Selecting the Hardware Target

5-4-7. Click **Next** to view the hardware target summary.

A summary of the connection is displayed.

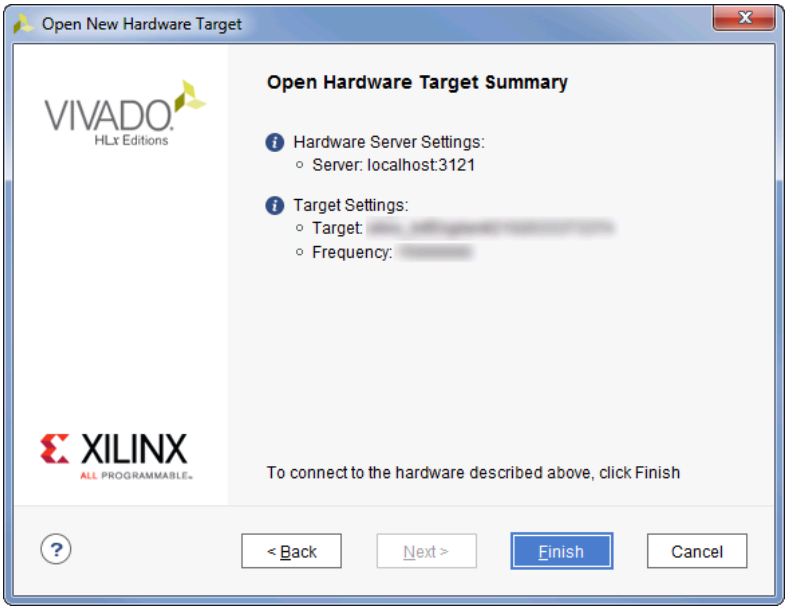


Figure 11-18: Summary of the Open Hardware Target Settings

5-4-8. Click **Finish** to connect to the new hardware target.

© Copyright 2019 Xilinx

5-5. Program the Kintex® UltraScale™ device using the hardware manager.

5-5-1. Select the **Hardware** tab.

5-5-2. Right-click **xcku040_0** and select **Program Device**.

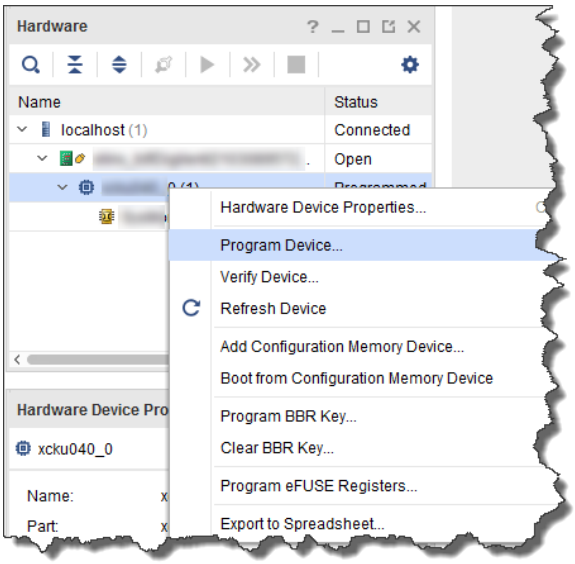


Figure 11-19: Programming the Device

The Program Device dialog box opens.

Observe that the bitstream file that will be used to program the device has been included in the Bitstream file field.

5-5-3. Click **Program**.

When the programming of the FPGA is complete, the status will show Programmed.

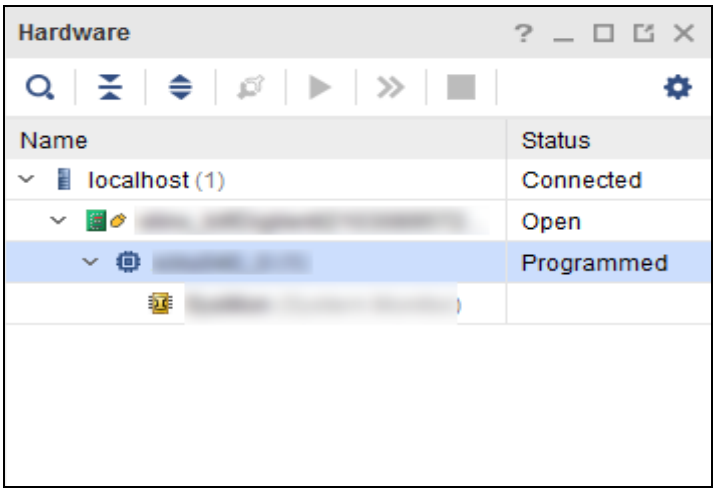


Figure 11-20: Programming Status of the FPGA

© Copyright 2019 Xilinx

Tera Term is a popular public domain terminal emulation program. It is capable of operating as a serial port terminal or as a telnet client. It is an alternative to using the Terminal view that is built into SDK.

The Tera Term setup file is provided in the `$flowLab/support` directory.

5-6. [Windows users only]: Launch the Tera Term terminal program.

5-6-1. Double-click the **Tera Term** icon on the Windows desktop to launch Tera Term.

Alternatively, you can select **Start** > **All Programs** > **Tera Term** > **Tera Term**.

5-6-2. Select **Serial** as the connection (1).

5-6-3. Click the **Port** drop-down list to view the available COM ports (2).

Note: If your port is not listed, exit Tera Term, power cycle your board and re-start this step.

5-6-4. Select the COM **#** discovered (3).

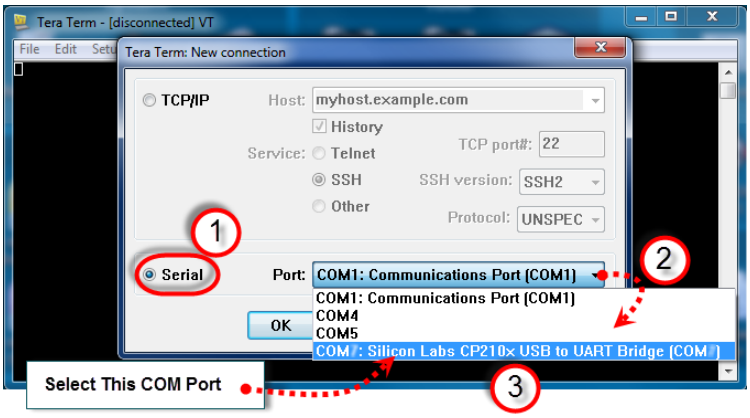


Figure 11-21: Selecting the COM Port

Note: The COM port setting is specific to the computer being used and may need to be different than shown. Use the COM port **#** that was discovered and specific to computer.

5-6-5. Click **OK**.

The terminal console window opens.

© Copyright 2019 Xilinx

5-6-6. Select **Setup** > **Serial Port**.

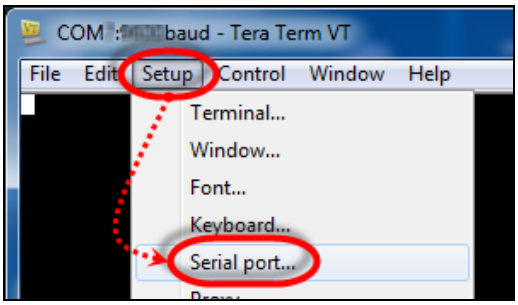


Figure 11-22: Opening the Tera Term Serial Port Setup Window

The Tera Term Serial Port Setup dialog box opens.

5-6-7. Confirm that the proper serial port has been selected (1).

5-6-8. Set the baud rate to **115200** (2).

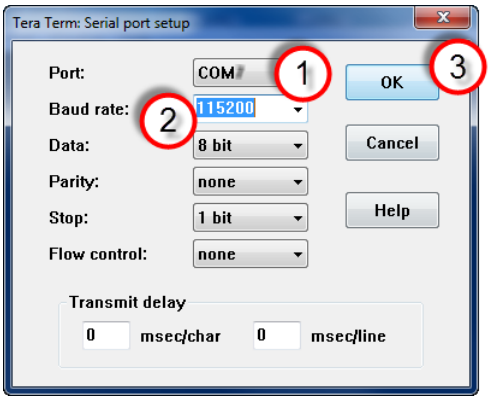


Figure 11-23: Setting the Parameters for the Serial Port

Note: The COM port setting is specific to the computer being used and may need to be different than shown. Use the COM port **#** that was discovered and specific to computer.

5-6-9. Click **OK** (3).

Tera Term is now configured to receive and transmit serial information to/from the evaluation board.

5-6-10. Select **Setup** > **Terminal** and select the **Local echo** option.

5-6-11. Click **OK**.

© Copyright 2019 Xilinx

5-7. [Linux users]: Launch GtkTerm and set the port configuration.

5-7-1. Press <**Ctrl** + **Alt** + **T**> launch a Linux terminal window.

5-7-2. Enter the following to launch GtkTerm from the terminal window:

```
sudo gtkterm
```

5-7-3. When prompted for the password, enter the superuser password.

VM users should use "**`xilinx`**".

Note: While the application will run as a regular user, you must be a super user to access the ports.

When the GtkTerm window opens, perform the following.

5-7-4. Select **Configuration** > **Port** to open the Configuration dialog box.

5-7-5. Identify the port associated with your board and set the port as **/dev/ttyUSBx** (where x could be 0, 1, 2, 3 etc.)

5-7-6. Set the baud rate to **115200**.

Leave the rest of the settings at their default.

Note: You can open multiple instances of **`/dev/USBx`** if you are unable to find out which port your UART is connected to.

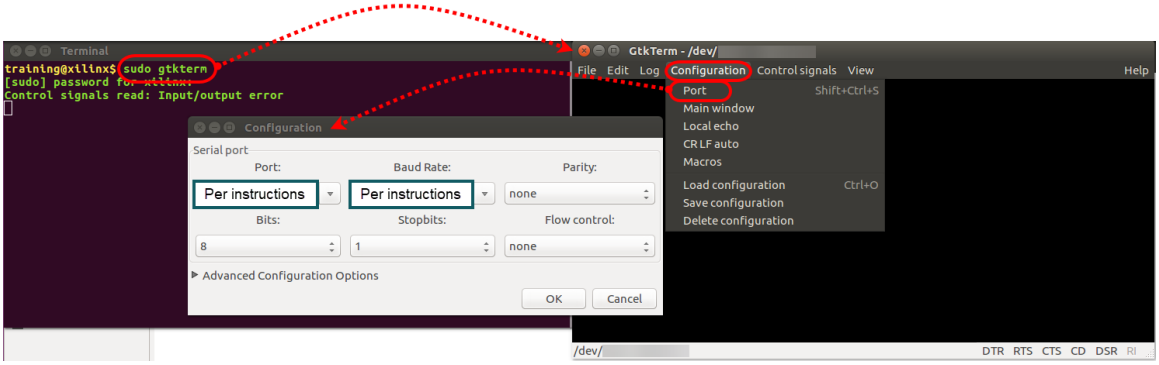


Figure 11-24: Opening GtkTerm and Selecting the Port Configuration

[Optional]: You can save these settings so that you do not have to reconfigure GtkTerm each time you open it by selecting **Configuration** > **Save configuration**.

If you save the configuration as "default" this configuration will open when GtkTerm starts. Otherwise you can save the configuration by another name; however, you will then need to load the configuration each time you start GtkTerm.

5-7-7. Click **OK** to save the settings and leave the terminal open.

© Copyright 2019 Xilinx

5-8. Enter data into the terminal and observe the pattern of the LEDs on the board.

5-8-1. Type any characters into the Tera Term window.

On the KCU105 board, all eight bits of the character will appear on the LEDs.

For a list of ASCII characters and their hexadecimal representation, see the *ascii_table.pdf* file in the `$flowLab/support` directory.

5-9. Close the hardware manager.

5-10. Power off the evaluation board.

5-11. Close Tera Term/GtkTerm.

5-12. Close the Vivado Design Suite.

[Optional]: Clean up the file system.

Some systems 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. Both processes will recursively delete all of the files in the `$flowLab` directory.

5-13. Clean up the VirtualBox file system via the GUI.

5-13-1. Using the graphical browser (Windows: press the <**Windows**> key + <**E**>; Linux: press <**Ctrl** + **N**>), navigate to `$flowLab`.

5-13-2. Select **`flowLab`**.

5-13-3. Press <**Delete**>.

5-14. Clean up the VirtualBox file system via the command line.

5-14-1. Open a terminal window (Windows: press the <**Windows**> key + <**R**>, then enter **cmd**; Linux: press <**Ctrl** + **Alt** + **T**>)

5-14-2. Enter the following command to delete the contents of the workspace:

[Windows users]: **`rd /s /q $flowLab`**

[Linux users]: **`rm -rf $flowLab`**

© Copyright 2019 Xilinx

Summary

In this lab, you completed the major stages of the Vivado tool design flow: creating a project, adding source files, simulating the design, synthesizing and implementing the design, and downloading the bitstream to the board.

© Copyright 2019 Xilinx

Answers

1. Why was running a simple simulation included in this lab?

   Behavioral simulation is an important element to include in the design process. Behavioral simulation verifies the functionality of the design. The Vivado Design Suite includes a simulator for this purpose.

© Copyright 2019 Xilinx