

Designing with SystemVerilog Lab Workbook

lang-svdes-2021.1-wkb-lab-rev1

Designing with SystemVerilog Lab Workbook 2021.1



© Copyright 2021 Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Kria, Spartan, Versal, Vitis, 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.html
 - These are original files and do not contain any work that you may have performed.
 - Labs were developed using version 2021.1 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).
 - These labs were validated using the Xilinx virtual machine which hosts Ubuntu Linux. Many labs can be performed using the Windows OS; however, not all Xilinx tools are supported under Windows (such as QEMU and PetaLinux).
- 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 considered a question for a bit, 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, Vitis platform, 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.
- Where can I find more information on lab software requirements and hardware setup?
 - The lab setup guide details this information and is available from the lab files download.
- How do the instructions work?
 - The instructions are provided in three 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 to which they are subordinate (X-X-X).

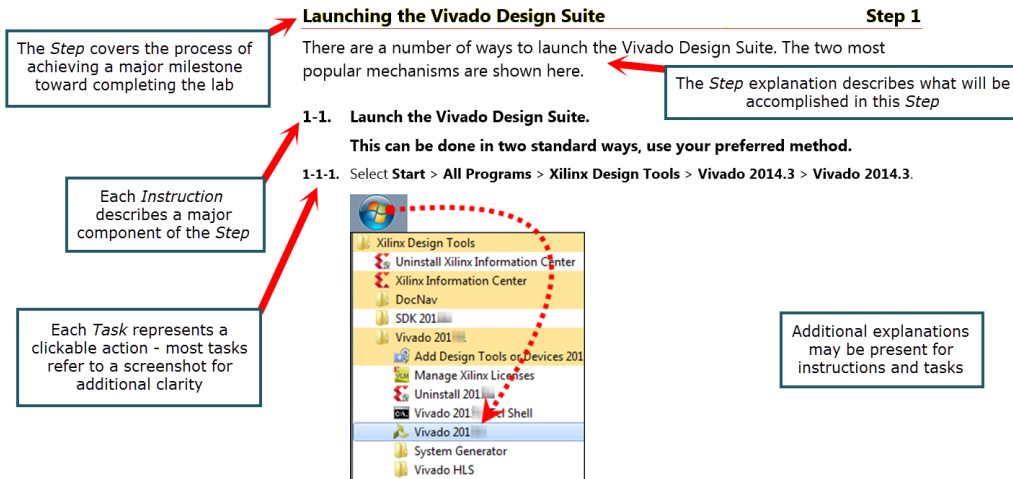


Figure 4-2: Launching the Vivado Design Suite from the Start Menu

-- OR --



Table of Contents

Lab 1: SystemVerilog Data Types.....	3
Lab 2: Structures	33
Lab 3: Unions.....	51
Lab 4: always_ff and always_comb Procedural Blocks.....	71
Lab 5: Functions, Tasks, and Packages	95
Lab 6: Interfaces and Design Download.....	115

Lab 1: SystemVerilog Data Types

2021.1

Abstract

This lab introduces the enum data type in SystemVerilog. You will perform two exercises in this lab:

- You will replace ``define` statements in a Verilog module with an *enum* type variable to illustrate the ease of using *enum* variables.
- You will edit a finite state machine (FSM) coded in Verilog to use the *enum* variable and synthesize the design in the Vivado® Design Suite.

This lab should take approximately 60 minutes.

Objectives

After completing this lab, you will be able to:

- Identify the benefits of *enum* data types and when to use them
- Simulate a SystemVerilog module
- Synthesize a SystemVerilog design using the Vivado Design Suite

Introduction

Enumerated Data Types

- Enumerated data types provide a means to declare an abstract variable that can have a specific list of valid values. The default data type for enum variables is *int* type.
- The enum element without a value assigned gets the increment of the previous element's value.
- Enum variables make code more readable when you are describing state machines and complex systems.

The security_1 FSM used in the second part of the lab is illustrated by the following state diagram:

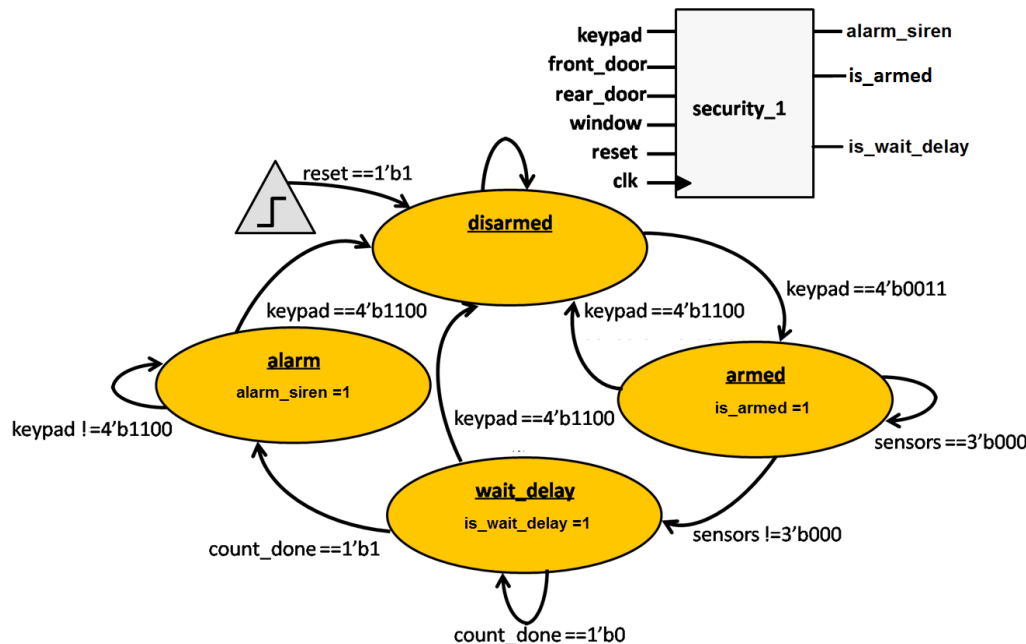


Figure 1-1: Block Diagram and State Diagram for Security System with Moore FSM

The signals in the above figure are modeled in the Verilog module as follows:

- keypad[3:0] – 4-bit input used to arm (0011) or disarm (1100) the security system.
- front_door, rear_door, window – Single-bit inputs which are assumed to go high when security is breached and the alarm should be activated.
- sensors[2:0] – 3-bit internal signal formed by concatenating the inputs - front_door, rear_door and window. If any of the 3 bits in this signal goes high, the alarm would be triggered after a delay of 100 clock cycles.
- clk – Master clock signal.
- reset – System reset.
- alarm_siren – Output which indicates that the system is in 'alarm' state (i.e, the alarm has been activated).
- is_armed – Output which indicates that the system is in 'armed' state (i.e, the security system is on).
- is_wait_delay – Output which indicates that the system is in 'wait_delay' state (i.e, a security breach has been detected and the system is waiting for 100 clock cycles before activating the alarm).
- start_count – Internal signal for counting 100 clock cycles before triggering the alarm.
- count_done – Internal signal that activates the alarm after 100 clock cycles have been counted.

The four states are declared as 2-bit constants using localparam.

There are three procedural blocks used to model the FSM:

- The first procedural block increments the state machine every clock cycle.
- The second procedural block determines what the next state should be.
- The third procedural block generates the output of the state machine.
- Additionally, there is a procedural block to count for 100 clock cycles before sounding the alarm.

Understanding the Lab Environment

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

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

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

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

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

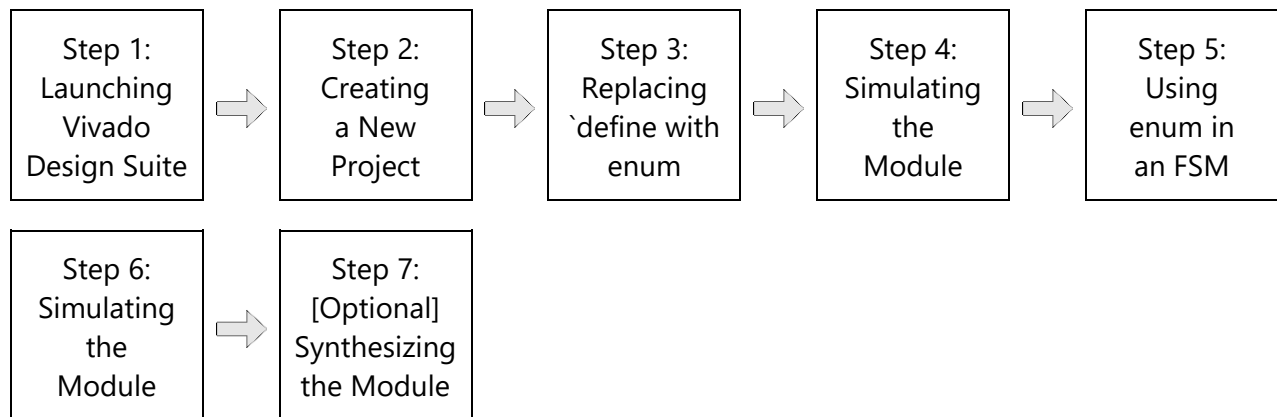
Environment Variable Name	Description
\$TRAINING_PATH	<p>Points to the space allocated for students to work through the labs. This directory includes prebuilt images and starting points for the labs and demos.</p> <p>The customer training VM sets \$TRAINING_PATH to the <code>/home/xilinx/training</code>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

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

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

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

General Flow



Launching the Vivado Design Suite

Step 1

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

1-1. Open the Vivado Design Suite.

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

OR

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

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

```
[host]$ vivado
```

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

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

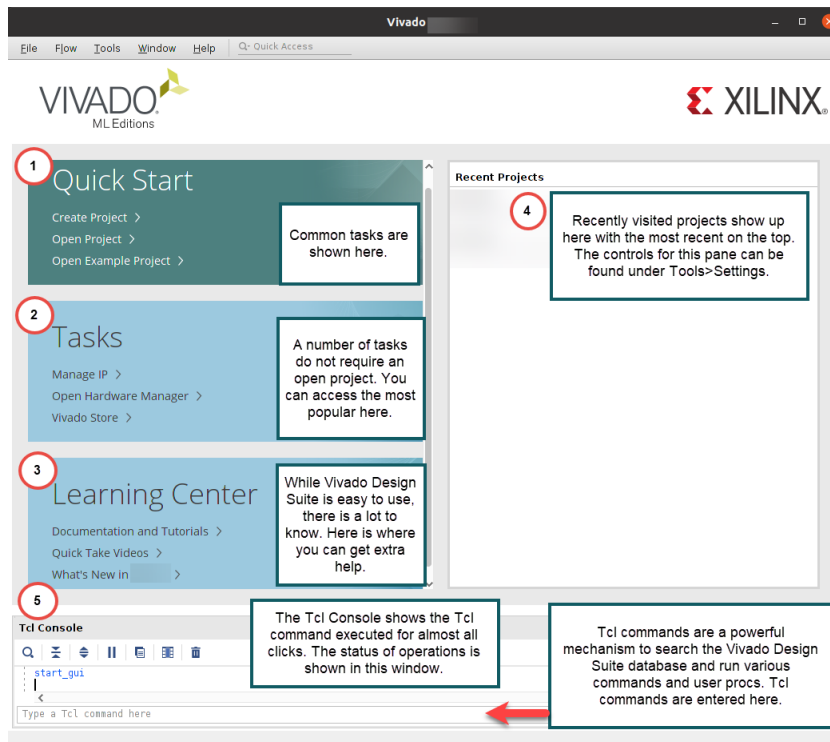


Figure 1-2: Vivado Design Suite Welcome Screen

Creating a New Vivado Design Suite Project

Step 2

"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.

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

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

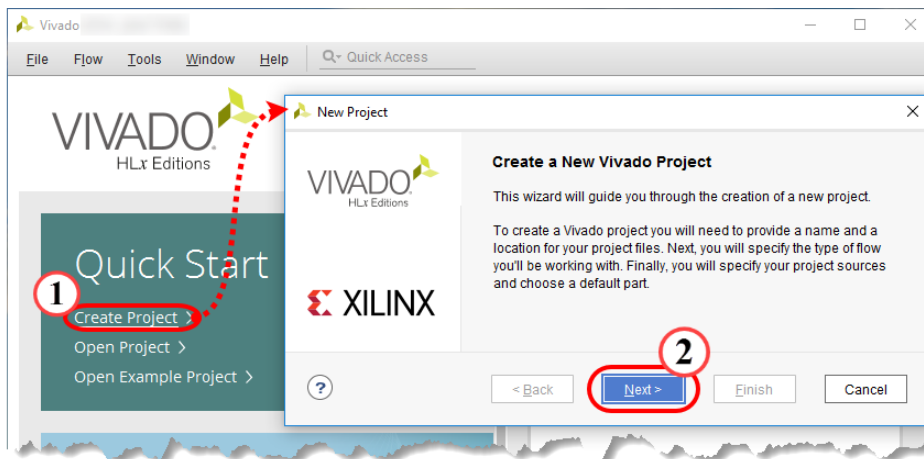


Figure 1-3: Creating a New Vivado Design Suite Project

This will launch the New Project Wizard.

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

2-2. Describe the various aspects of the project.

2-2-1. Enter **sv_data_types** in the Project name field (1).

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

```
$TRAINING_PATH/sv_data_types/lab/KCU105
```

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

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

2-2-3. Deselect the **Create Project Subdirectory** option (3).

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

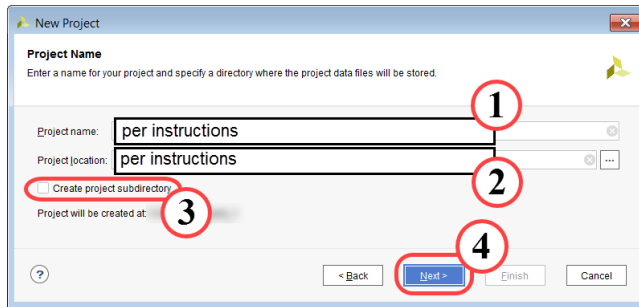


Figure 1-4: Entering the Project Name and Location

2-2-4. 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.

2-2-5. Select **RTL Project**.

2-2-6. Select **Do not specify sources at this time**, which creates a blank project.

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

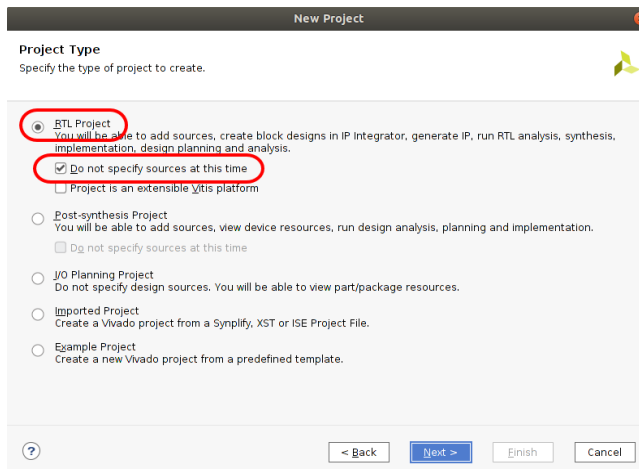


Figure 1-5: Specifying Project Options

2-2-7. Click **Next** to advance to the target device/platform selection.

2-3. 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.

2-3-1. Click **Boards** from the *Default part* area to filter by board rather than by the specific part (1).

2-3-2. 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.

2-3-3. Select **Kintex-UltraScale KCU105 Evaluation Platform** from the board list.

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

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

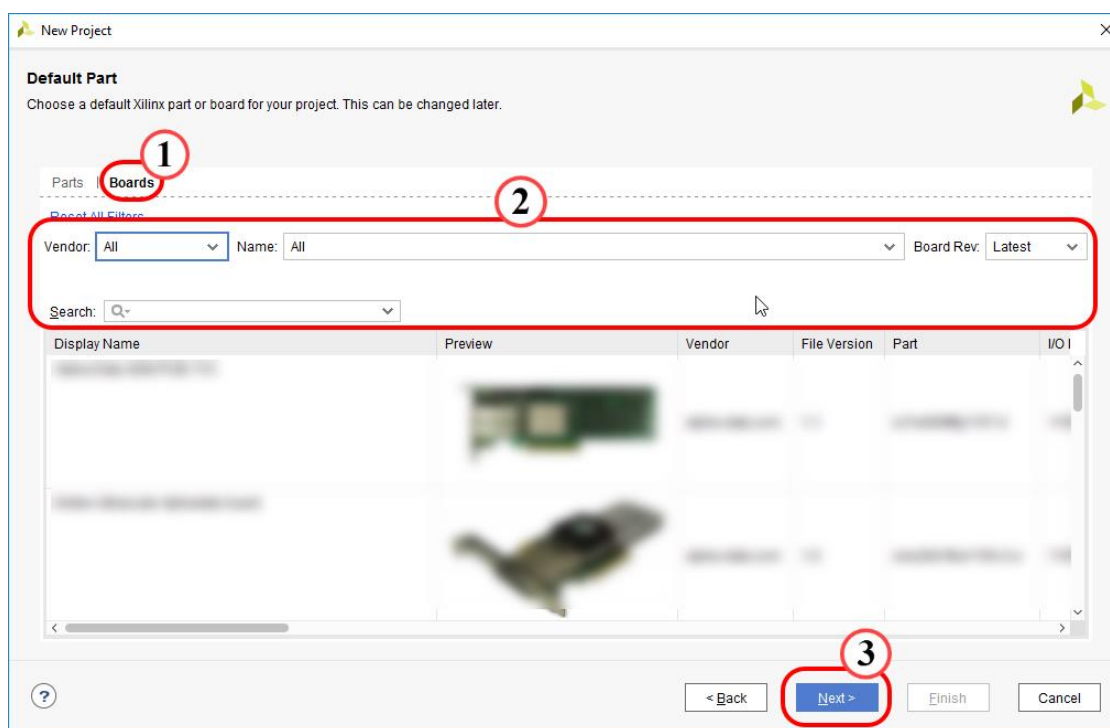


Figure 1-6: Selecting the Board for the Project

2-3-4. Click **Next** to advance to the summary (3).

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

2-3-5. 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.

Replacing ``define` Statement Variables with `enum`

Step 3

You will edit a Verilog module that prints the ASCII value of each character in a word. The 26 ``define` statements will be replaced with a single `enum` variable in SystemVerilog.

3-1. Open and analyze an existing Verilog module.

- 3-1-1. Select **File > Text Editor > Open File** in the Project window.
- 3-1-2. Browse to the `$sv_data_types/support/src_files` directory.
- 3-1-3. Select **print_ascii_verilog**.
- 3-1-4. Click **OK**.
- 3-1-5. Observe that each letter in the alphabet uses a ``define` statement to hold its corresponding ASCII value.

In this module, the word whose characters need to be displayed in ASCII is hard-coded as "hello".

Hence, a register array called `word` is declared to have five array elements (one for each character), each element having a size of 8 bits (size of ASCII value).

Notice that the variable `lcd_bus` is used to hold the ASCII value of each character within a `for` loop.

3-2. Create a new SystemVerilog file called `print_ascii_sv`.

- 3-2-1. Select **Add Sources** in the Flow Navigator, under Project Manager.

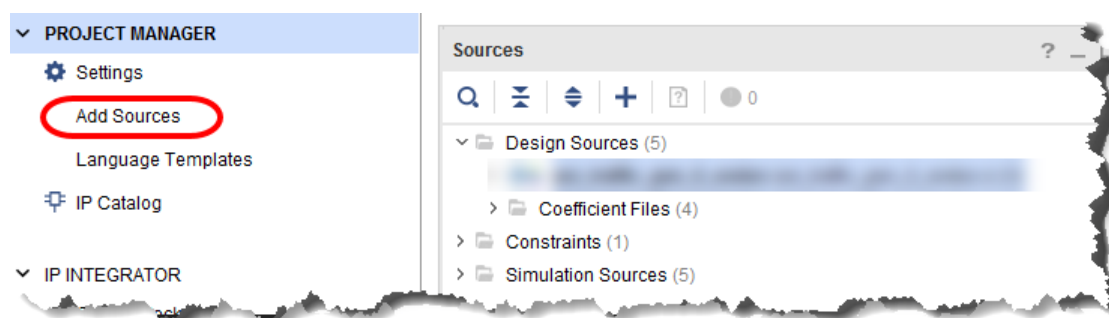
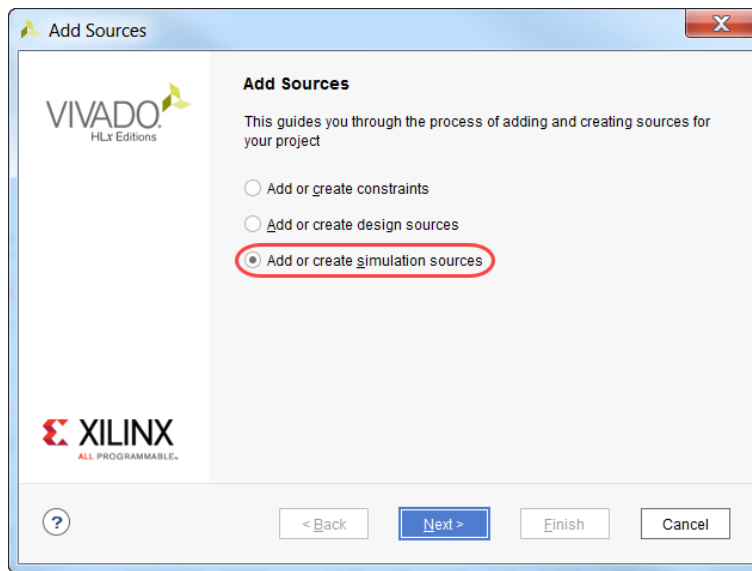


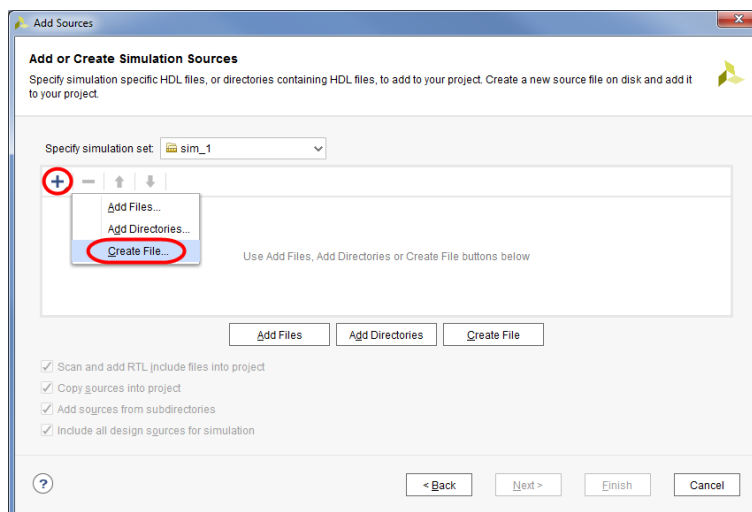
Figure 1-7: Selecting Add Sources

3-2-2. Select Add or create simulation sources.**Figure 1-8: Add Sources Dialog Box**

Note that selecting **Add or create simulation sources** will designate the file for simulation only. When creating a design source file, you would select **Add or create design sources**, which will designate the file for both synthesis and simulation.

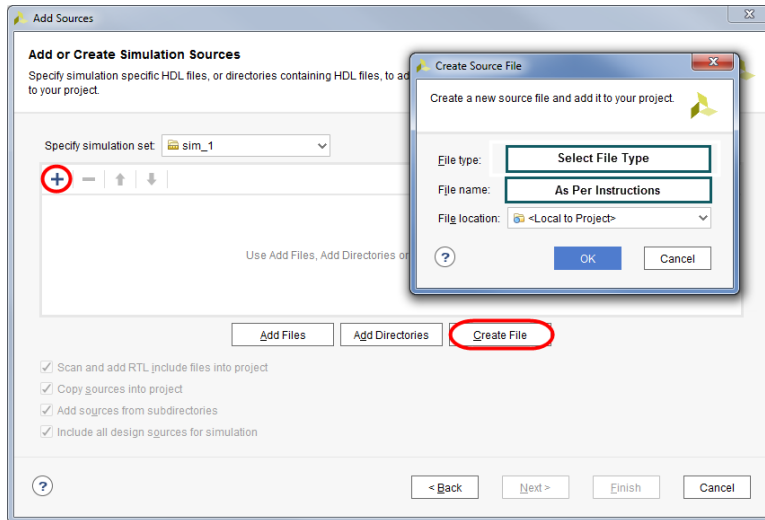
3-2-3. Click Next.

The Add or Create Simulation Sources dialog box opens.

3-2-4. Click the Plus (+) icon and select Create File.**Figure 1-9: Selecting Create File**

The Create Source File dialog box opens.

3-2-5. Enter print_ascii_sv as the file name.

3-2-6. Select **SystemVerilog from the File type drop-down list.****Figure 1-10: Entering Testbench Filename and Type****3-2-7. Click **OK** in the Create Source File dialog box.****3-2-8. Click **Finish**.**

The Define Module dialog box opens.

3-2-9. Click **OK to create the module without ports, since this module does not require them.****3-2-10. Click **Yes** to confirm that there the module definition has not been changed.****3-3. Open the newly created SystemVerilog file for editing.****3-3-1. Expand **Simulation Sources** > **sim_1** in the Sources window.****3-3-2. Double-click **print_ascii_sv.sv**.**

The SystemVerilog file opens for editing.

3-4. Copy the Verilog code to the newly created SystemVerilog file.**3-4-1. Select the **print_ascii_verilog.v** tab to view the file.****3-4-2. Right-click and select **Select All**.****3-4-3. Right-click and select **Copy**.****3-4-4. Select the **print_ascii_sv.sv** tab.****3-4-5. Select the contents of the tab (<Ctrl + A>), and press <Delete> to clear the contents of **print_ascii_sv.sv**.****3-4-6. Right-click and select **Paste**.**

3-5. Edit the SystemVerilog file.

3-5-1. Rename the module to **print_ascii_sv**.

3-5-2. Replace the 26 ``define` statements with a single *enum* variable.

Hint: Group all 26 letters into one enum variable called *alphabet*. Use a *bit* data type with size [7:0] for each element of the *enum* variable. Recall that in *enum* an element will be initialized to the preceding element's value plus 1. Hence, if consecutive elements in an *enum* have to be incremented by 1, then it is sufficient to initialize only the first element of the *enum*.

You can refer to the ASCII table located in the `support` directory (`$sv_data_types/support/ascii_table.pdf`) to verify the ASCII value of each letter.

3-5-3. Replace the `'define` statements with *enum* variables, such as `'h` with *h*, `'e` with *e*, and so on.

3-5-4. Replace the *reg* type variables with *logic* and update the word variable to a packed array as shown in the Answer section.

Considering that logic types can be used to represent both variables and wires, it is a convenient coding practice to use *logic* data types, especially for large designs.

3-5-5. Remove the declaration of the *cnt* variable.

Unlike in Verilog, in SystemVerilog, variables that control *for* loops can be declared within the *for* loop. This enhances the readability of the code in addition to ease of coding.

3-5-6. Update the *for* loop to include the declaration of *cnt* as a *bit* type variable with a size of 3 bits.

```
for(bit [2:0] cnt=0; cnt<5; cnt=cnt+1)
```

3-5-7. Select **File > Text Editor > Save** to save the file.

Note: The completed code is available for your reference in the Answers section.

Simulating the `print_ascii_sv` Module

Step 4

You will simulate the `print_ascii_sv` module and observe the outputs.

The following instructions are for Vivado® simulator users only. Mentor Graphics Questa users can proceed to the instruction that begins with "The following instructions are for Questa users only".

4-1. Run the simulation.

- 4-1-1.** Select **Simulation > Run Simulation > Run Behavioral Simulation** from the Flow Navigator.

The simulation runs for the default 1 us.

The ASCII values of each of the five characters in the word "hello" will be displayed in the Tcl Console.

4-2. Verify the simulation result.

- 4-2-1.** Use the ASCII table in the `support` directory (`$sv_data_types/support/ascii_table.pdf`) to verify that the ASCII value displayed in the TCL Console for each character is correct.

4-3. Exit the simulator.

- 4-3-1.** Select **File > Close Simulation**.
- 4-3-2.** Click **OK** to confirm.

The following instructions are for **Questa** users only. **Vivado** simulator users can proceed to the next step of this lab.

4-4. Set the simulation target to Questa Advanced Simulator.

- 4-4-1. Select **Settings** under Project Manager from the Flow Navigator.
- 4-4-2. Set the Target Simulator field to **Questa Advanced Simulator** under Simulation.
- 4-4-3. Click **Yes** in the Target Simulation dialog box.

This changes the target simulator to Questa Advanced Simulator.

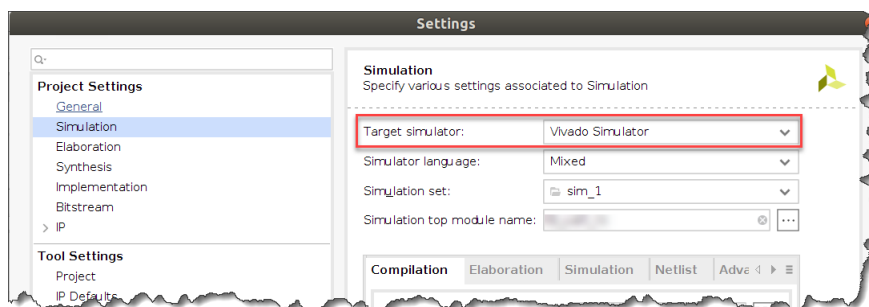


Figure 1-11: Simulation Settings

- 4-4-4. Ensure the Simulator language is set to Verilog.
- 4-4-5. Click **OK** in the **Settings** dialog box.

4-5. Run the simulation.

- 4-5-1. Select **Simulation** > **Run Simulation** > **Run Behavioral Simulation** from the Flow Navigator.

The ASCII values of each of the five characters in the word "hello" will be displayed in the transcript window.

4-6. Verify the simulation result.

- 4-6-1. Use the ASCII table in the `support` directory (`$sv_data_types/support/ascii_table.pdf`) to verify that the ASCII value displayed for each character is correct.

4-7. Exit the simulator.

- 4-7-1. Select **File** > **Quit**.
- 4-7-2. Click **Yes** to confirm.

Using enum for Modeling in a Finite State Machine

Step 5

You will add and edit an existing Verilog module that models a finite state machine that represents a security system. The digital logic used to represent the states will be replaced with the *enum* variable, making it easier to code, debug, and understand the module.

5-1. Add an HDL source file to the design.

5-1-1. Select **Add Sources** from the Flow Navigator tab, under Project Manager.

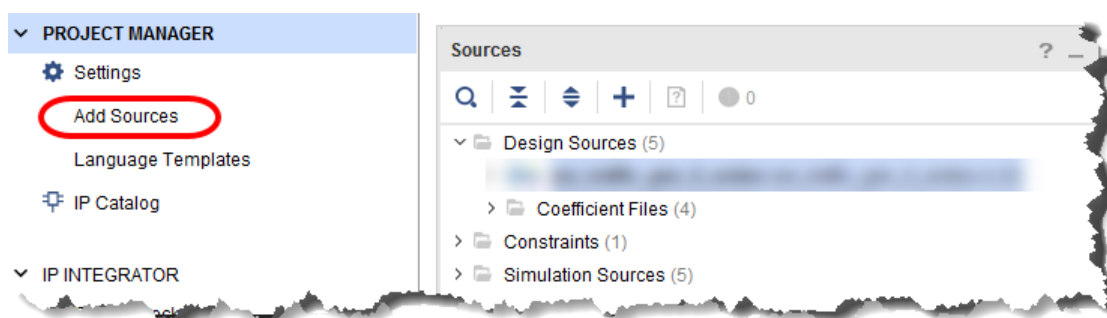


Figure 1-12: Selecting Add Sources

The Add Sources dialog box opens, allowing you to add HDL source files to the project.

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

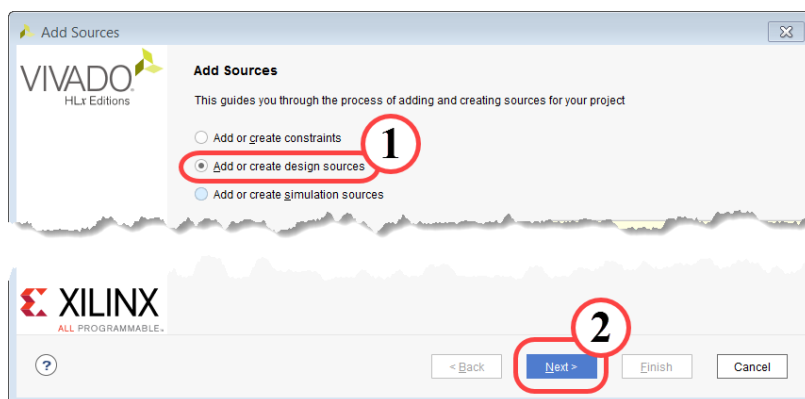


Figure 1-13: Selecting Add or Create Design Sources

5-1-3. Click **Next** to begin selecting source files.

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.

5-1-4. Click the **Plus (+)** icon and select **Add Files**.

5-1-5. Browse to \$sv_data_types/support/src_files.

5-1-6. Select **security_verilog.v** file.

5-1-7. Click **OK** in the Add Source Files dialog box to select the file(s).

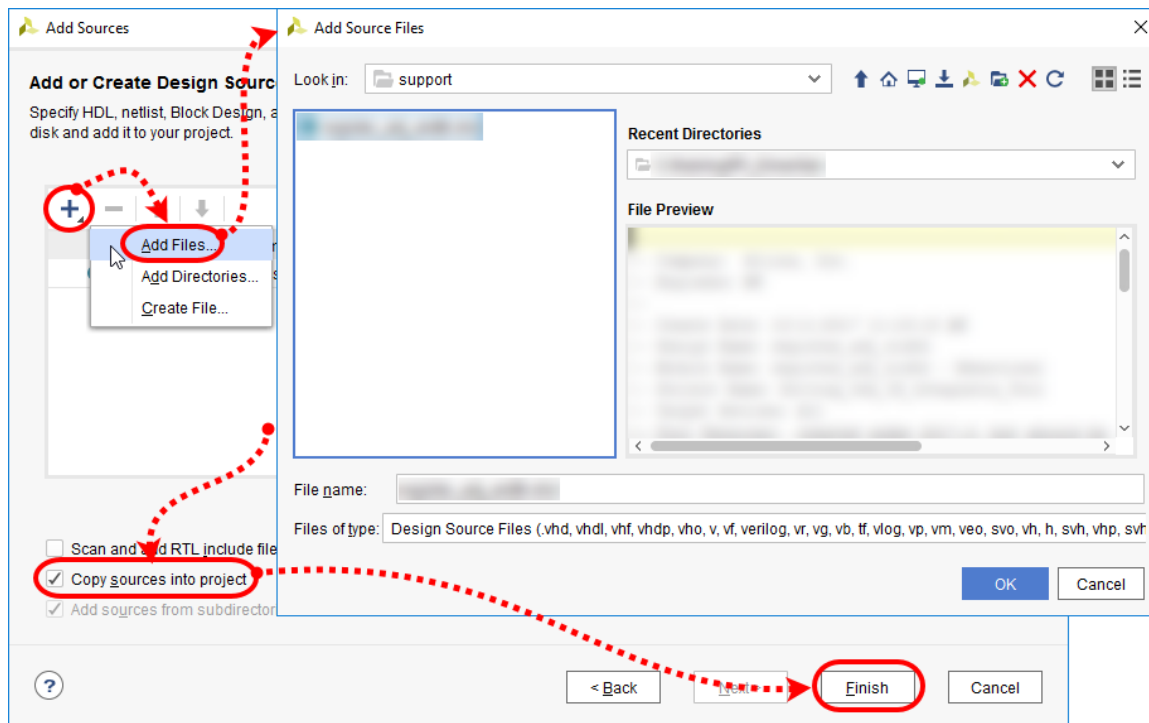


Figure 1-14: Selecting Add Files

5-1-8. Ensure that the **Copy sources into project** option is selected.

5-1-9. Click **Finish** in the Add or Create Design Sources dialog box to add the HDL sources to the project.

5-2. Open and analyze the Verilog module.

5-2-1. Expand **Design Sources** and double-click **security_verilog.v** in the Sources window.
The Verilog file opens.

5-2-2. Verify that the logic matches the description given in the Introduction section.

5-3. Create a new HDL source file called *security_sv*.

5-3-1. Select **Add Sources** in the Flow Navigator under Project Manager.

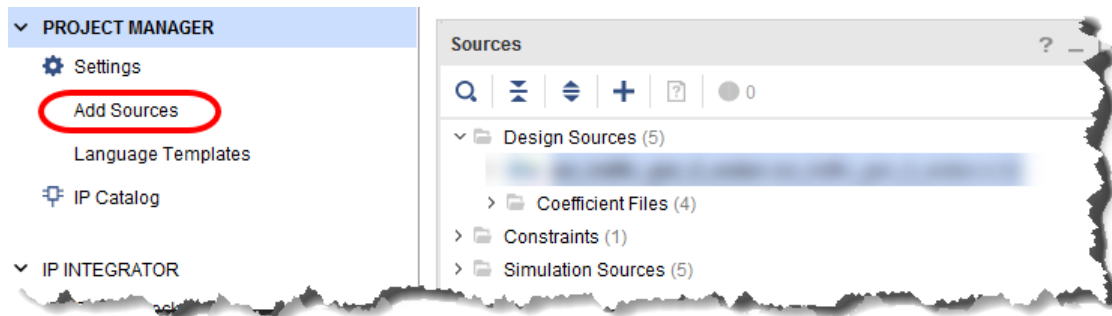


Figure 1-15: Selecting Add Sources

5-3-2. Select **Add or create design sources** (1).

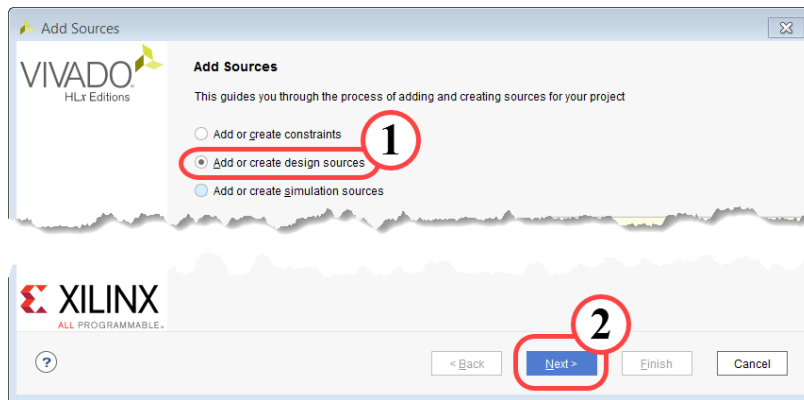


Figure 1-16: Selecting Add or Create Design Sources

5-3-3. Click **Next** (2).

The Add or Create Design Sources dialog box opens.

5-3-4. Click the **Plus (+)** icon and select **Create File**.

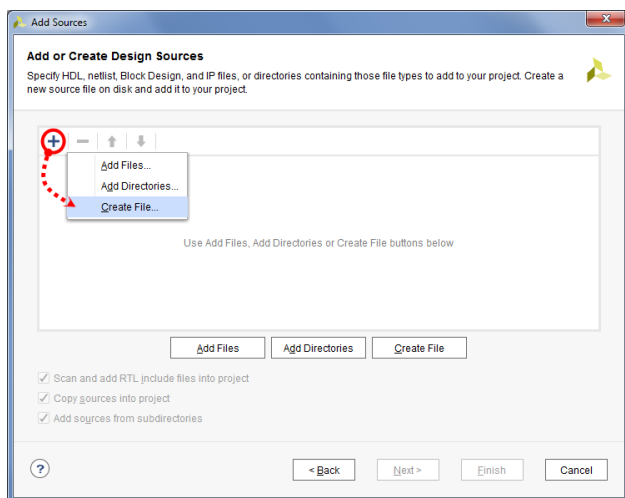


Figure 1-17: Selecting Create File

The Create Source File dialog box opens.

5-3-5. Select **SystemVerilog** from the File type drop-down list.

5-3-6. Enter **security_sv** as the file name.

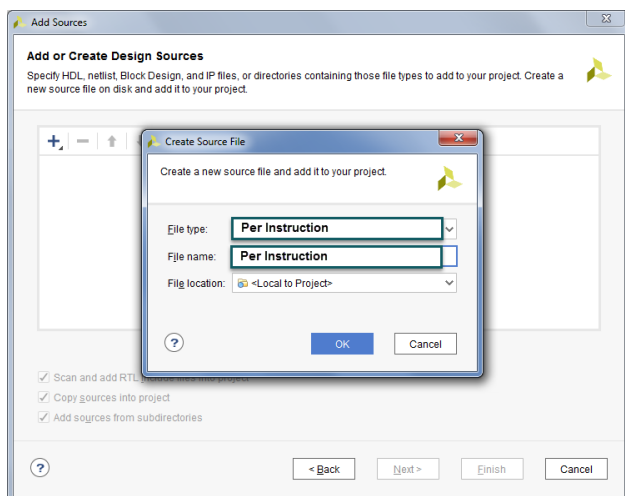


Figure 1-18: Entering File Name and Type

5-3-7. Click **OK** in the Create Source File dialog box.

5-3-8. Click **Finish** to add the new source file(s).

The Define Module dialog box opens.

5-3-9. Click **OK** to create the module without ports.

You will use the contents of the *security_verilog* module to create this module.

5-3-10. Click **Yes** to confirm that the module definition has not been changed.

5-4. Open the newly created SystemVerilog file for editing.

5-4-1. Expand **Design Sources** from the Sources window.

5-4-2. Double-click **security_sv.sv**.

The SystemVerilog file opens for editing.

5-5. Copy the Verilog code to the newly created SystemVerilog file.

5-5-1. Select the **security_verilog.v** tab to view the file.

5-5-2. Right-click and select **Select All**.

5-5-3. Right-click and select **Copy**.

5-5-4. Select the **security_sv.sv** tab.

5-5-5. Select all (<Ctrl + A>), and press <Delete> to clear the contents of **security_sv.sv**.

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

5-6. Edit the SystemVerilog file.

5-6-1. Rename the module as **security_sv**.

5-6-2. Replace the *reg* type output ports with *logic* type output ports.

5-6-3. Replace the *reg* type and wire type variables with the *logic* type.

5-6-4. Remove the declaration of states as local parameters.

5-6-5. Remove the declaration of *current_state* and *next_state* variables that represent one of the four states.

In the Verilog code, *current_state* and *next_state* variables represent one of the four states by holding a 2-bit value corresponding to one of the four states.

5-6-6. Use the *enum* type to represent the four states using labels instead of using logic values.

This enables easier coding, debugging, and understanding of the finite state machine.

Hint: Each of *current_state* and *next_state* variables represent any one of the four states at a given time. Hence, the elements of the enum should be the state names. The enum variables should be *current_state* and *next_state*.

5-6-7. Select **File > Text Editor > Save File** to save the file.

Note: The completed code is available for your reference in the Answers section.

Simulating the security_sv Module

Step 6

You will simulate the *security_sv* module by using a testbench (*security_sv_tb*) that is provided. You will observe the outputs and verify that the FSM code works correctly.

HDL simulation files can be added to the design at any time.

6-1. Add simulation files to the design.

6-1-1. Select **Add Sources** under Project Manager in the Flow Navigator.

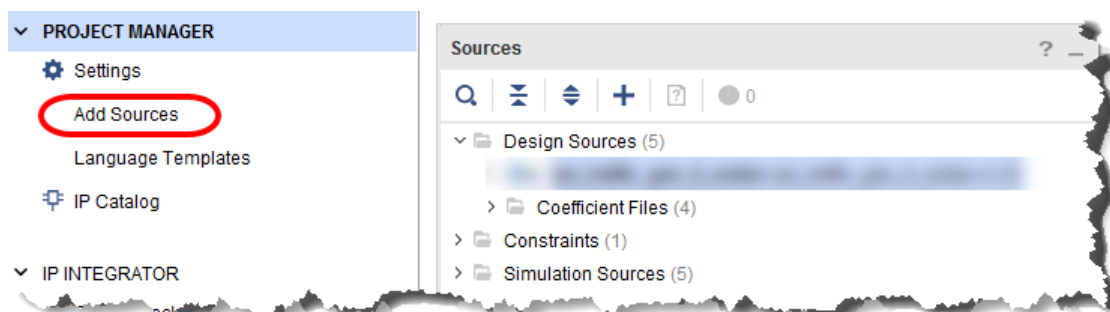


Figure 1-19: Selecting Add Sources

6-1-2. Select **Add or create simulation sources**.

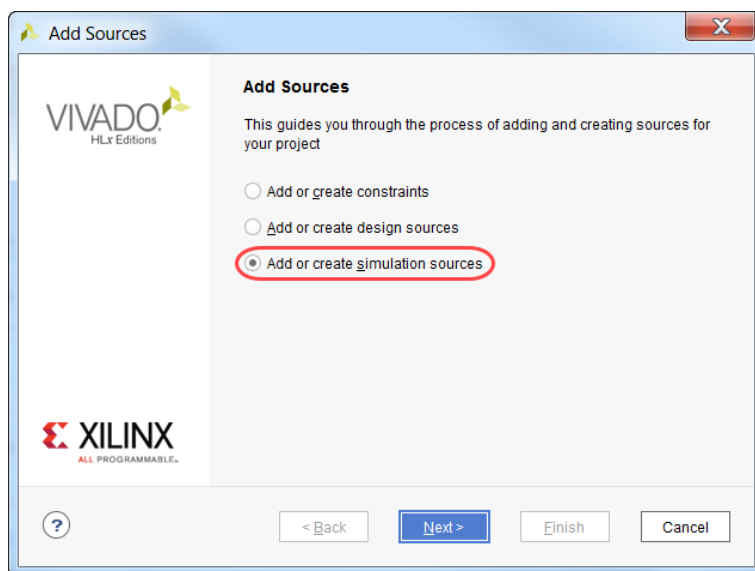


Figure 1-20: Add Sources Dialog Box

6-1-3. Click **Next**.

6-1-4. Click the **Plus (+)** icon to open the pop-up menu.

6-1-5. Select **Add Files** to open the Add Source Files dialog box which allows you to browse to the desired directory.

- 6-1-6. Browse to the `$TRAINING_PATH/sv_data_types/support` directory if it is not open already.
- 6-1-7. Select **security_sv_tb.sv**.
- 6-1-8. Click **OK** in the Add Source Files dialog box.
- 6-1-9. Ensure that the **Copy sources into project** option is selected.
- 6-1-10. Click **Finish** to add the file(s) to the project.

6-2. Open and analyze the provided testbench.

- 6-2-1. Expand **Simulation Sources** > **sim_1** in the Sources window.

Note how the Vivado Design Suite detects that *security_sv* is part of the *security_sv_tb* module and organizes it accordingly under Simulation Sources.

- 6-2-2. Double-click **security_sv_tb.sv**.

The SystemVerilog testbench opens.

- 6-2-3. Analyze the following attributes of the testbench:

- The function *check_state* is defined to verify if each of the three outputs of the module (*security_sv*) is at its expected state. If not, issue a suitable error message.
- The *initial* block at the end of the module defines a specific sequence of conditions that verifies if the system enters each state, remains at that state, and transitions to other states as defined in the FSM description.
- Display "test passed" if the module (*security_sv*) passes all the checks.

- 6-2-4. Close the **security_sv_tb.sv** file.

The following instructions are for Vivado simulator users only. Mentor Graphics Questa users can proceed to the instruction that begins with "The following instructions are for Questa users only".

Users who are working on both simulators will need to change the simulation settings to the Vivado simulator.

6-3. Run the simulation.

- 6-3-1. Ensure that **security_sv_tb.sv** is set as the top file.

If it is not, right-click **security_sv_tb.sv** in the Sources window and select **Set as Top**.

- 6-3-2. Select **Simulation** > **Run Simulation** > **Run Behavioral Simulation** from the Flow Navigator.

The simulation runs for the default 10 us.

6-4. Add signals that are present lower in the hierarchy and are of interest.

6-4-1. Select **security_sv_tb** > **uut** in the Scopes pane.

You can now view the UUT instance that is the lower-level module (*security_sv*) relative to the testbench.

The Simulation Objects pane now shows the signals at the UUT level of hierarchy.

6-4-2. Drag-and-drop the **curr_state** and any other signal of interest to the waveform window.

6-5. Change the radix of the signals as necessary.

6-5-1. Right-click the signal keypad in the waveform window and select **Radix** > **Binary**.

This will make it easier to analyze the waveform. You can similarly change the radix of any other signal of interest.

6-6. Restart and rerun the simulation in interactive mode.

6-6-1. Select **Run** > **Restart** to restart the simulation.

You can also click the Restart icon from the toolbar.



Figure 1-21: Simulation Restart and Run Options

6-6-2. Select **Run** > **Run For**.

6-6-3. Enter **50** as the time in the Run For dialog box.

Make sure **microseconds** is selected from the units drop-down box.

6-6-4. Click **OK** to start the simulation.

This will fully validate the transitions described in the testbench.

Note that the simulation will end once all tests have been cleared. You should see "test passed" displayed in the Tcl Console window.

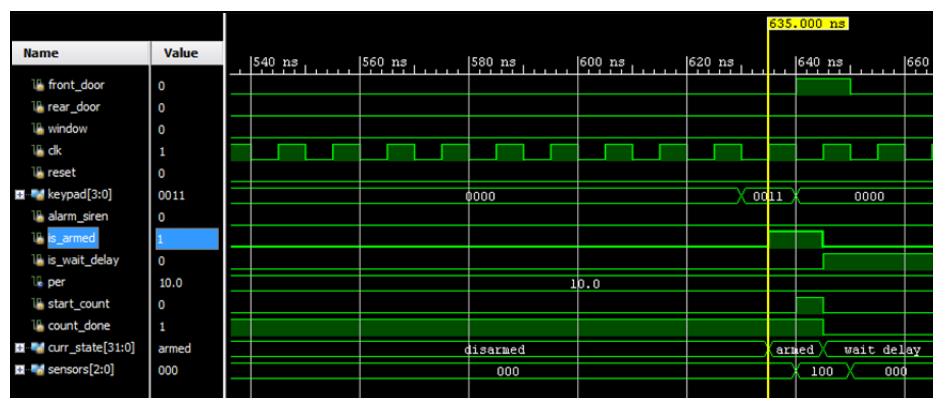


Figure 1-22: Simulation Results

- 6-6-5. Use the zoom options in the toolbar to scale the waveform window.
- 6-6-6. Observe the waveforms corresponding to signals such as *clk*, *keypad*, *sensors*, *curr_state*, *start_count*, etc. to validate the complete functionality of the FSM.

6-7. Exit the simulator.

- 6-7-1. Select **File > Close Simulation**.
- 6-7-2. Click **OK** to confirm.
- 6-7-3. Click **Discard** to exit without saving the waveform.

The following instructions are for Questa users only. Vivado simulator users can proceed to the next step of this lab.

6-8. Run the simulation.

- 6-8-1. Ensure that **security_sv_tb.sv** is set as the top file.
If this is not, right-click **security_sv_tb.sv** in the Sources window and select **Set as Top**.
- 6-8-2. Select **Settings > Simulation** from the Flow Navigator and ensure that Target Simulator is set to Questa Advanced Simulator.
- 6-8-3. Select **Simulation > Run Simulation > Run Behavioral Simulation** from the Flow Navigator.
This invokes the Questa Sim and opens the simulator with the default wave window.

6-9. Add signals whose waveforms need to be analyzed.

- 6-9-1. Select **View > Objects** if the Objects window is not opened.
- 6-9-2. Select **security_sv_tb > uut** in the sim-Default window.
All the top-level signals of the unit under test (*security_sv*) will be displayed in the Objects window.
- 6-9-3. Select signals of interest in the Object window.
- 6-9-4. Right-click the selected signals and select **Add Wave**.
The selected signals will be added to the default wave window.

6-10. Change the radix of the signals as necessary.

- 6-10-1. Right-click the signal **keypad** in the waveform window and select **Radix > Binary**.
This will make it easier to analyze the waveform. You can similarly change the radix of *delay_val* and *start_count* to **Decimal**.

6-11. Run the simulation for 50 us.**6-11-1.** Select **Simulate > Restart****6-11-2.** Click **OK** to restart simulation.**6-11-3.** Type **run 50us** in the Transcript window and press **<Enter>**.**6-11-4.** Click **No** when prompted if you want the simulation to finish.

This will bring up the waveform for viewing.

The "test passed" comment should be displayed in the Transcript window along with other comments corresponding to each state.

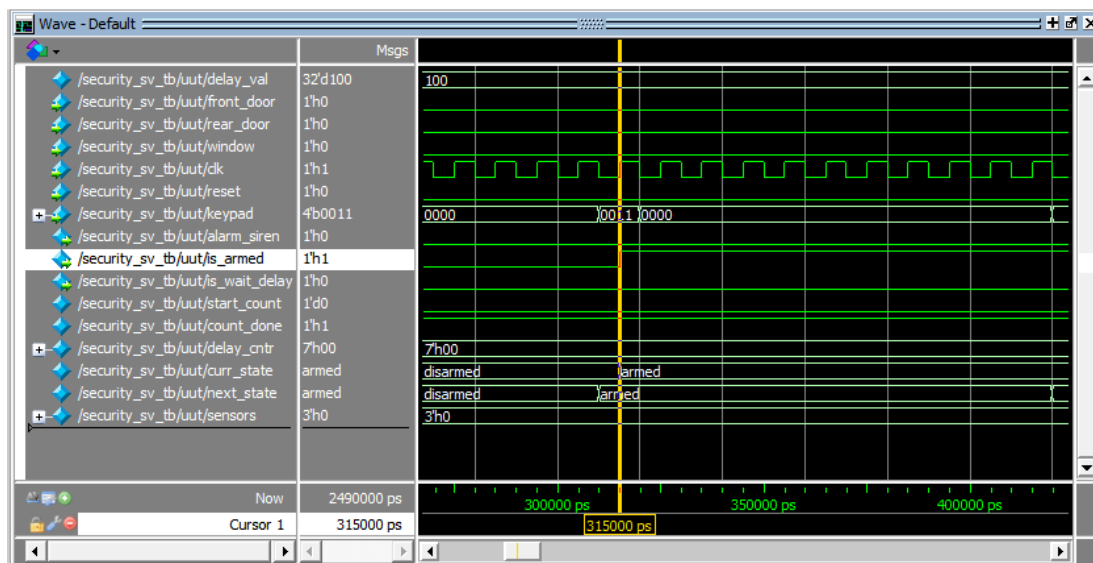
6-12. Verify the simulation result.**6-12-1.** Select the **Wave** tab to view the waveforms.**6-12-2.** Use the zoom options in the toolbar to scale the waveform window.

Figure 1-23: Simulation Results Showing Outputs Being in Sync with curr_state

6-12-3. Observe the waveforms corresponding to signals such as *clk*, *keypad*, *sensors*, *curr_state*, *start_count*, etc., to validate the complete functionality of the FSM.

6-13. Exit the simulator.**6-13-1.** Select **Simulate > End Simulation** to close the simulator.**6-13-2.** Click **Yes** to confirm.**6-14. Close Questa Sim.****6-14-1.** Select **File > Quit**.**6-14-2.** Click **Yes** to close Questa Sim.

Optional: Synthesizing the `security_sv` Module

Step 7

In this step, you will synthesize the `security_sv` design to show that the enum data type is supported by Vivado Synthesis tool.

7-1. Set `security_sv` as top module if not already set.

7-1-1. Right-click `security_sv` under Design sources and select **Set as Top**.

7-2. Explore the synthesis settings available.

7-2-1. Select **Settings > Synthesis** in the Flow Navigator.

Note that the strategy is set to **Vivado Synthesis Defaults**.

7-2-2. Review the other synthesis options.

For this lab, you will **not** modify the implementation options. Nevertheless, you can view the currently available implementation options by selecting **Implementation**.

7-2-3. After reviewing the other options, click **OK**.

7-3. Run design synthesis based on the settings selected.

7-3-1. Click **Run Synthesis** in the Flow Navigator.

7-3-2. Click **OK**.

You can view any of the reports generated after synthesis by selecting **View Reports** in the Synthesis Completed dialog box.

7-4. Close the Vivado Design Suite.

7-4-1. Select **File > Exit**.

The Exit Vivado dialog box opens.



Figure 1-24: Exit Vivado Dialog Box

7-4-2. Click **OK**.

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

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

7-5. [Optional] [Only for local VMs—not for CloudShare] Clean up the file system.

Using the GUI:

7-5-1. Using the graphical browser (Windows: press the **<Windows>** key + **<E>**; Linux: press **<Ctrl + N>**), navigate to `$TRAINING_PATH/sv_data_types`.

7-5-2. Select **sv_data_types**.

7-5-3. Press **<Delete>**.

-- OR --

Using the command line:

7-5-4. Open a terminal window (Windows: press the **<Windows>** key + **<R>**, then enter **cmd**; Linux: press **<Ctrl + Alt + T>**).

7-5-5. Enter the following command to delete the contents of the workspace:

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

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

Summary

In this lab, you learned how to create a new project in the Vivado Design Suite.

Next, you learned how the *enum* type variable can be used to simplify initialization of variables when the values held by the variables need to be incremented by 1 sequentially. You also saw the benefits of using *enum* type variables in modeling finite state machines.

Finally, you learned how to synthesize a SystemVerilog design in the Vivado Design Suite.

Answers

print_ascii_sv.sv

```
module print_ascii_sv;

    //Use enum to simplify the assignment of each letter
    enum bit [7:0] {a=8'h61, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z} alphabet;

    logic [0:4] [7:0] word; // Use logic type and update word to a packed array
    logic [7:0] lcd_bus; //Use logic type for lcd_bus

    initial
    begin
        word[0] = h;
        word[1] = e;
        word[2] = l;
        word[3] = l;
        word[4] = o;

        for(bit [2:0] cnt=0; cnt<5; cnt=cnt+1) // declare cnt in the for loop
        begin
            lcd_bus = word[cnt];
            $display ("ASCII value of word[%d] = %h", cnt, lcd_bus);
        end

    end

endmodule
```

security_sv.sv

```
`timescale 1ns / 1ps

module security_sv(
    input front_door,
    input rear_door,
    input window,
    input clk,
    input reset,
    input [3:0] keypad,
    output logic alarm_siren, // replace all reg and wire with logic
    output logic is_armed,
    output logic is_wait_delay
);

    // set the delay value (the number of clocks between a faulted zone and the
    // alarm going off)

    parameter          delay_val      = 100;
```

```
// Variables used for counting 100 (delay_val) clock cycles
logic start_count;
logic count_done;
logic [6:0] delay_cntr = 0 ; // Max value of delay_cntr is delay_val (i.e., d'100 or b'1100100)

// Enumerated types provide a means for defining a variable that has a
//   restricted set of legal values. The values are represented with labels
// instead of digital logic values.
enum {disarmed, armed, wait_delay, alarm} curr_state, next_state;

logic [2:0] sensors ; // used to combine inputs
assign sensors = { front_door, rear_door, window } ;

// procedural block for incrementing the state machine
always @ ( posedge clk )
    if (reset)
        curr_state <= disarmed ;
    else
        curr_state <= next_state ;

// procedural block to determine the next state
always @ ( curr_state, sensors, keypad, count_done ) begin
    case ( curr_state )
        disarmed: begin
            if ( keypad == 4'b0011 )
                next_state <= armed;
            else
                next_state <= curr_state ;
        end
        armed: begin
            if ( sensors != 3'b000 )
                next_state <= wait_delay;
            else if ( keypad == 4'b1100 )
                next_state <= disarmed;
            else
                next_state <= curr_state ;
        end
        wait_delay: begin
            if (count_done == 1'b1)
                next_state <= alarm;
            else if ( keypad == 4'b1100 )
                next_state <= disarmed ;
            else
                next_state <= curr_state ;
        end
    endcase
end
```

```

        next_state <= curr_state ;
    end
    alarm: begin
        if ( keypad == 4'b1100 )
            next_state <= disarmed;
        else
            next_state <= curr_state ;
        end
    endcase
end

// procedural block to generate the state machine output values
always @ ( posedge clk ) begin
    if (reset) begin
        is_armed      <= 1'b0 ;
        is_wait_delay <= 1'b0 ;
        alarm_siren   <= 1'b0 ;
    end
    else
    begin
        is_armed      <= ( next_state == armed );
        is_wait_delay <= ( next_state == wait_delay );
        alarm_siren   <= ( next_state == alarm );
    end
end

assign start_count = (( curr_state == armed) && (sensors != 3'b000));

// Implement the delay counter.
// Loads delay_cntr with delay_val-1 when start_count is high, then counts down to 0 and stops.
// The condition delay_cntr = 0 triggers the next state transition in the main state machine

always @ ( posedge clk) begin
    if (reset)
        delay_cntr <= 0;
    else if (start_count)
        delay_cntr <= delay_val - 1'b1;
    else if (curr_state != wait_delay)
        delay_cntr <= 0;
    else if (delay_cntr != 0)
        delay_cntr <= delay_cntr - 1'b1;
    end

    assign count_done = (delay_cntr == 0);
endmodule

```


Lab 2: Structures

2021.1

Abstract

This lab introduces structures in SystemVerilog.

This lab should take approximately 30 minutes.

Objectives

After completing this lab, you will be able to:

- Write code for a structure in SystemVerilog
- Simulate and synthesize a SystemVerilog design using the Vivado® Design Suite

Introduction

The arith_logic SystemVerilog model used in this lab works as a miniature 8-bit ALU. It performs four arithmetic or four logical operations on 8-bit operands.

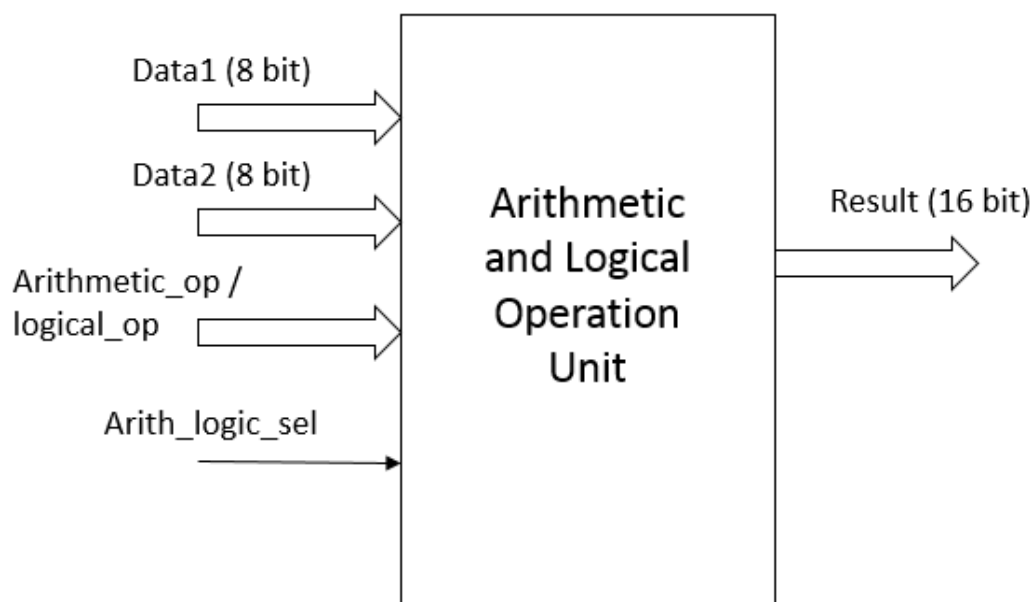


Figure 2-1: Block Diagram

The enumerated type variable `arithmetic_op` chooses between the four arithmetic operations that are available: addition, subtraction, multiplication, and division. Similarly, the `logical_op` variable decides between the four logical operations that are available: `nand`, `nor`, `not`, and `xor`.

These operations are done on two data signals, which are specified in the packed structure called `arith_logic_info`. The final result is of 16 bits.

The `arith_logic_sel` signal selects between the arithmetic and logical operations.

You will find the following signals in this lab:

- `arith_logic_sel` – Selects between arithmetic and logical operations.
- `operation` – Selects which arithmetic or logical operation is to be done.
- `ip_data1` and `ip_data2` – Two 8-bit data inputs.
- `data_out` – Output of the selected operation.

Apart from these, there are two objects created in this module, through which the structures and their members can be accessed:

- `arith_data` – Object of the packed structure `arith_logic_info` performing arithmetic operations.
- `logic_data` – Object of the packed structure `arith_logic_info` performing logical operations.

Understanding the Lab Environment

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

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

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

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

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

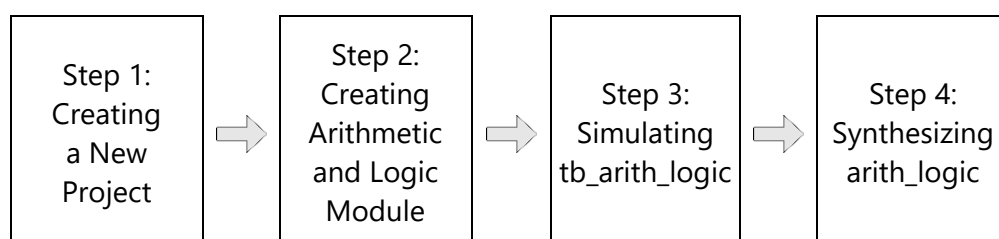
Environment Variable Name	Description
\$TRAINING_PATH	<p>Points to the space allocated for students to work through the labs. This directory includes prebuilt images and starting points for the labs and demos.</p> <p>The customer training VM sets \$TRAINING_PATH to the <code>/home/xilinx/training</code>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

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

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

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

General Flow



Creating a New Vivado Design Suite Project

Step 1

You will launch the Vivado Design Suite and create a new project.

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*.

"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).

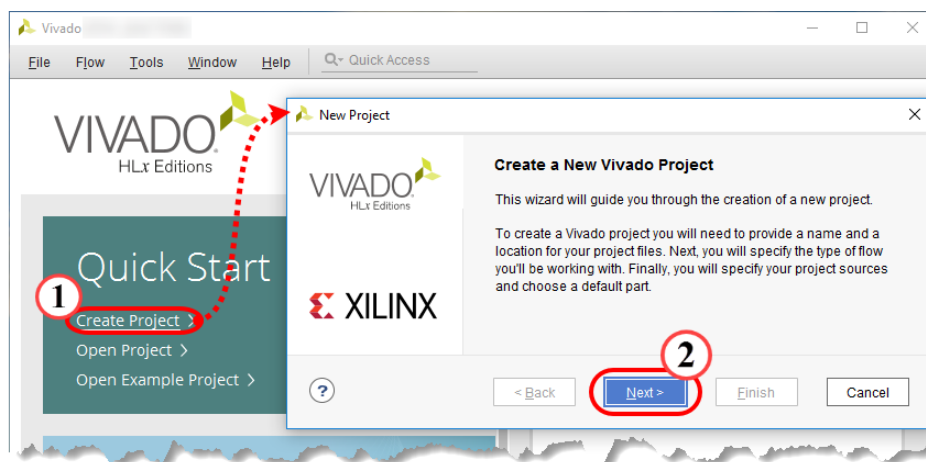


Figure 2-2: Creating a New Vivado Design Suite Project

This will launch the New Project Wizard.

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

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

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

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

```
$TRAINING_PATH/sv_struct/lab/KCU105
```

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

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

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

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

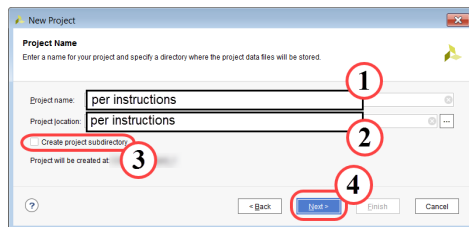


Figure 2-3: Entering the Project Name and Location

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

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

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

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

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

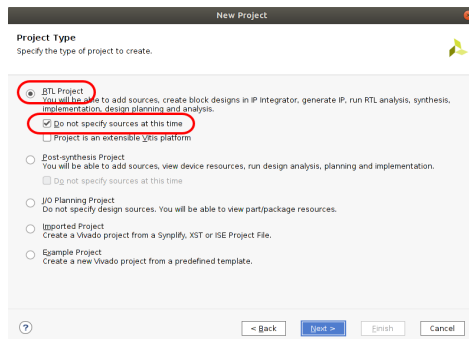


Figure 2-4: Specifying Project Options

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

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

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

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

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

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

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

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

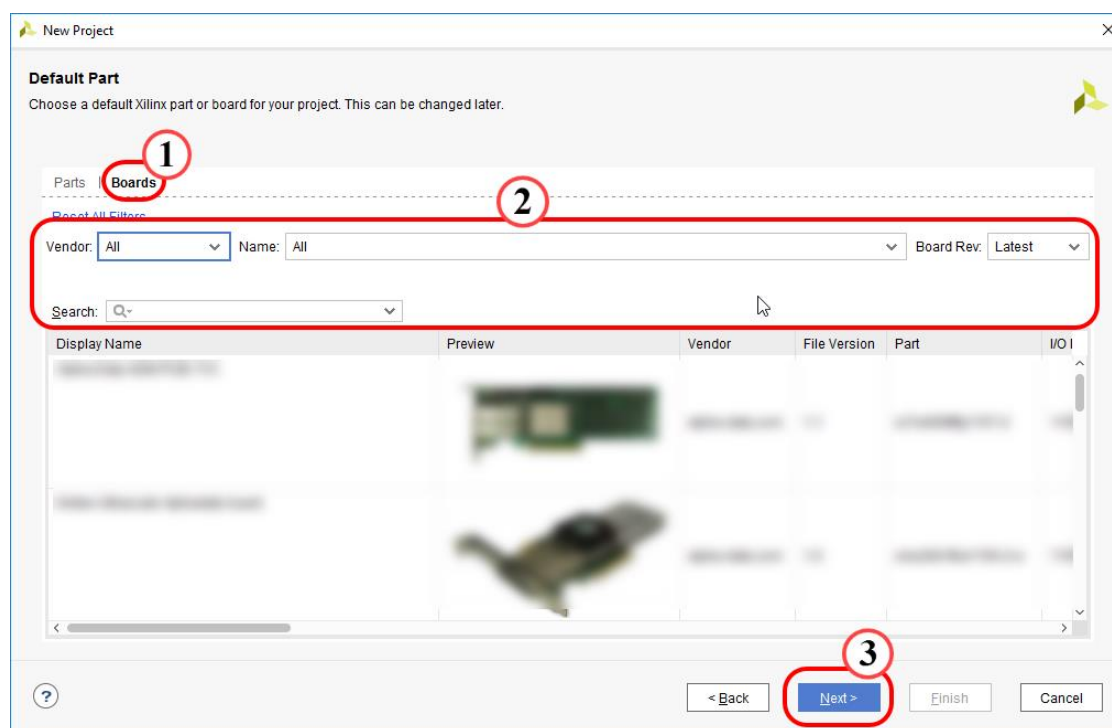


Figure 2-5: Selecting the Board for the Project

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

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

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

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

Creating the Arithmetic and Logic Module

Step 2

In this step, you will edit a SystemVerilog module that will perform arithmetic or logical operations, depending upon the input given by the user.

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

2-1-1. Select **Add Sources** from the Flow Navigator tab, under Project Manager.

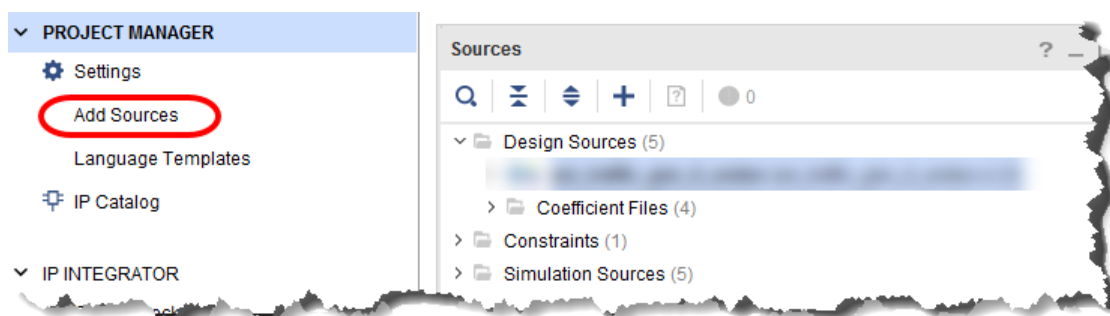


Figure 2-6: Selecting Add Sources

The Add Sources dialog box opens, allowing you to add HDL source files to the project.

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

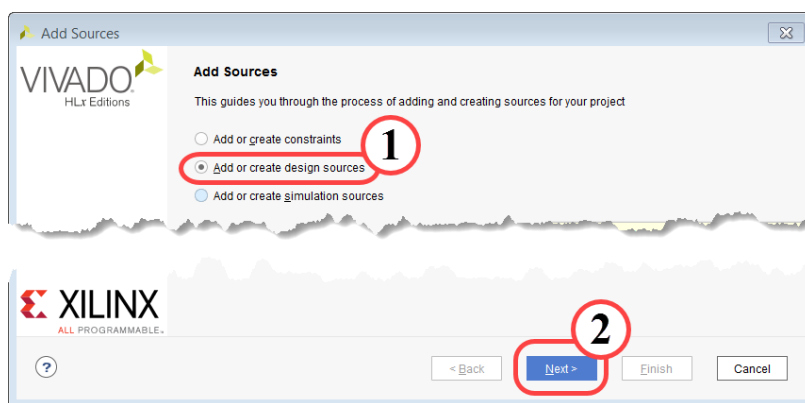


Figure 2-7: Selecting Add or Create Design Sources

2-1-3. Click **Next** to begin selecting source files.

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 `$sv_struct/support/src`.

2-1-6. Select **arith_logic**.

2-1-7. Click **OK** in the Add Source Files dialog box to select the file(s).

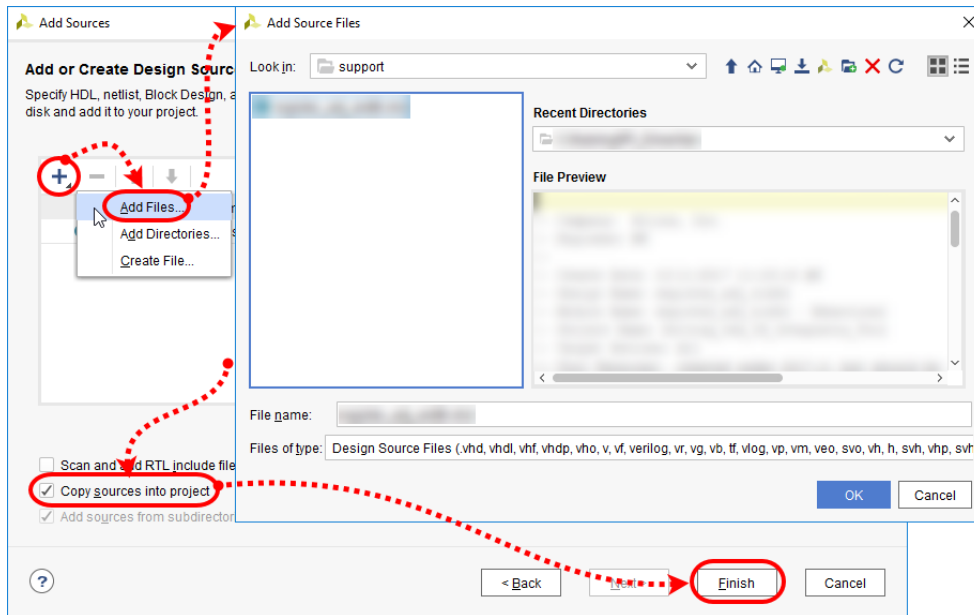


Figure 2-8: Selecting Add Files

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. Open and edit an existing SystemVerilog module.

2-2-1. Expand **design sources > arith_logic** in the Sources window.

2-2-2. Double-click to open the file.

Observe that the enumerated type variables `arith_operation` and `logic_operation` are declared at the start. This module performs two kinds of operations: arithmetic or logical.

You can select between them with the help of the `arith_logic_sel` signal mentioned in the *if-else* statement.

2-2-3. Locate the comment "Creating one Packed and one Unpacked Structure here" near line 27.

2-2-4. Create a user-defined packed structure.

This structure should have six members: `arithematic_op`, `logical_op`, `data1`, `data2`, `arith_result`, and `logic_result`. `data1` is of the logic data type and `data2` is of byte.

The other two members are of the enum type, declared as `arith_operation`, `logic_operation` earlier.

2-2-5. Now code the rest of the section as below:

- Locate the comment "Creating two objects arith_data, logic_data for arith_logic_info" near line 45. Create two objects arith_data and logic_data for struct arith_logic_info.
- Locate the comment "Selecting Arithmetic Operation" near line no. 48. Observe the code to check if arithmetic operation is to be done.
- Locate the comment "Read the input data" near line no. 51. Observe the code to read what operation needs to be done. Store it in arith_data member arithmetic_op. Read the inputs ip_data1 and ip_data2 and store it in arith_data members data1 and data2.
- Locate the comment "As per the value of enum, do the arithmetic operations". Observe the code to check what operation (add, sub, mul and div) is to be done using a *case* statement. Use the struct arith_data members to perform this operation.
- Locate the comment "Write the result to the output" near line no. 69. Observe the computed output to the data_out port.
- Similarly, observe the code for the logical operation.
- Select **File > Text Editor > Save File** to save the file.

Note: Refer to the Answers section at the end of this lab to verify the code.

2-3. Adding a new SystemVerilog testbench file called tb_arith_logic.

HDL simulation files can be added to the design at any time.

2-3-1. Select **Add Sources** under Project Manager in the Flow Navigator.

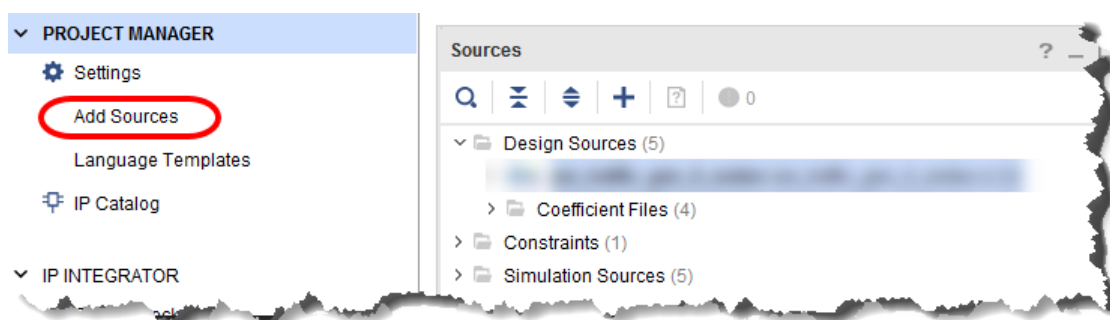


Figure 2-9: Selecting Add Sources

2-3-2. Select **Add or create simulation sources**.

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

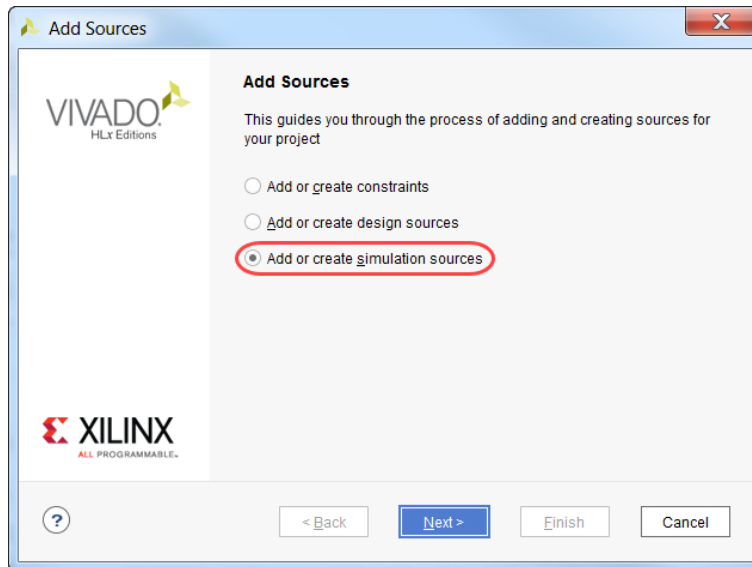


Figure 2-10: Add Sources Dialog Box

2-3-4. Click the **Plus (+)** icon to open the pop-up menu.

2-3-5. Select **Add Files** to open the Add Source Files dialog box which allows you to browse to the desired directory.

2-3-6. Browse to the `$TRAINING_PATH/sv_struct/support/src` directory if it is not open already.

2-3-7. Select **tb_arith_logic.sv**.

2-3-8. Click **OK** in the Add Source Files dialog box.

2-3-9. Click **Finish** to add the file(s) to the project.

2-3-10. Observe the SystemVerilog testbench file to see how the initialization is done.

2-3-11. Observe how the stimulus is given to the module ports.

Simulating the tb_arith_logic Module

Step 3

In this step, you will simulate the tb_arith_logic module and observe the outputs.

The following instructions are for Vivado simulator users only. Mentor Graphics Questa users can proceed to the instruction that begins with "The following instructions are for Questa users only".

3-1. Run the simulation.

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

The simulation runs for the default **1us**.

3-1-2. Use the zoom options in the toolbar to scale the waveform window.

3-2. Change the radix and verify the simulation result.

3-2-1. Verify that the arith_logic_sel signal decides between the arithmetic and logical operation in the waveform window.

3-2-2. Right-click the ip_data1, ip_data2, and data_out signals and select **Radix > Unsigned Decimal**.

This will make it easier to analyze the waveform while checking the arithmetic operations (arith_logic_sel = 0).

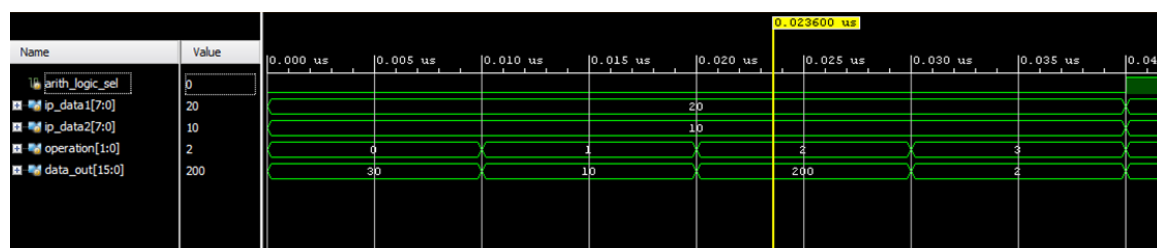


Figure 2-11: Simulation Result Showing Arithmetic Operation

The operation signal decides between the arithmetic operations, with operation = 0 being add, operation = 1 being subtract, and so on. Verify the data_out value as per the operation selected.

3-2-3. Right-click the ip_data1, ip_data2, and data_out signals again and select **Radix > Binary**.

This will make it easier to analyze the waveform while checking the logical operations (arith_logic_sel = 1).

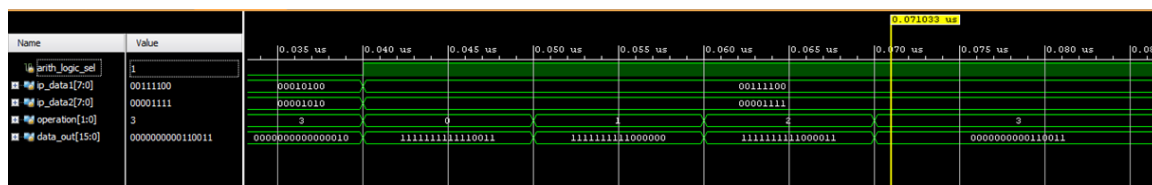


Figure 2-12: Simulation Result Showing Logical Operation

This time, the operation signal decides between the logical operations, with operation = 0 being nand_op, operation = 1 being nor_op, and so on. Verify the data_out value as per the operation selected.

3-3. Exit the simulator.

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

3-3-2. Click **OK** to confirm.

3-3-3. Click **Discard** to exit without saving the waveform.

The following instructions are for Questa users only. Vivado simulator users can proceed to the next step of this lab.

3-4. Set the simulation target to Questa Advanced Simulator.

3-4-1. Select **Settings** under Project Manager from the Flow Navigator.

3-4-2. Set the Target Simulator field to **Questa Advanced Simulator** under Simulation.

3-4-3. Click **Yes** in the Target Simulation dialog box.

This change the target simulator to Questa Advanced Simulator.

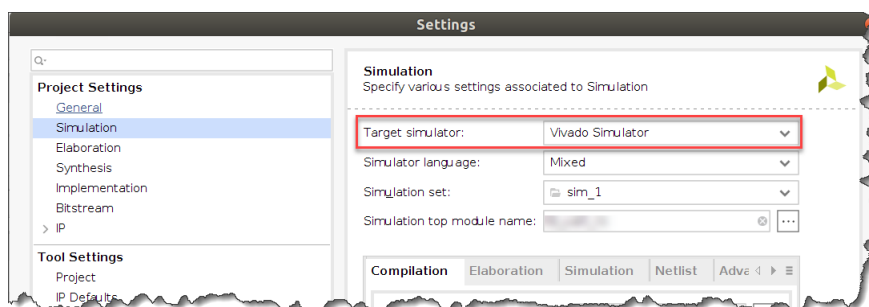


Figure 2-13: Simulation Settings

3-4-4. Click **OK** in the Settings dialog box.

3-5. Run the simulation.

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

3-5-2. Use the zoom options in the toolbar to scale the waveform window.

3-6. Change the radix and verify the simulation result.

3-6-1. Verify that arith_logic_sel signal decides between the arithmetic and logical operations in the waveform window.

3-6-2. Right-click the ip_data1, ip_data2, and data_out signals and select **Radix > Decimal**.

This will make it easier to analyze the waveform while checking arithmetic operations (arith_logic_sel = 0).

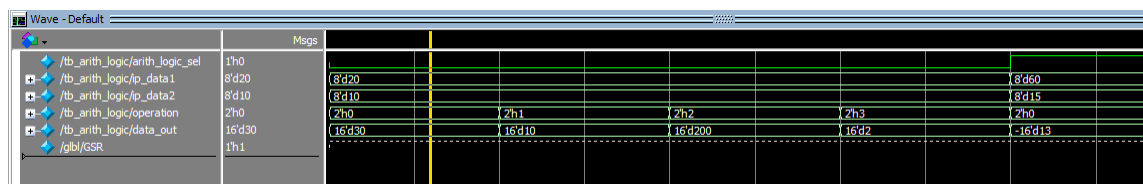


Figure 2-14: Simulation Results

The operation signal decides between the arithmetic operations, with operation = 0 being add, operation = 1 being subtract, and so on. Verify the data_out value as per the operation selected.

3-6-3. Right-click the ip_data1, ip_data2, and data_out signals again and select **Radix > Binary**.

This will make it easier to analyze the waveform while checking logical operations (arith_logic_sel = 1).

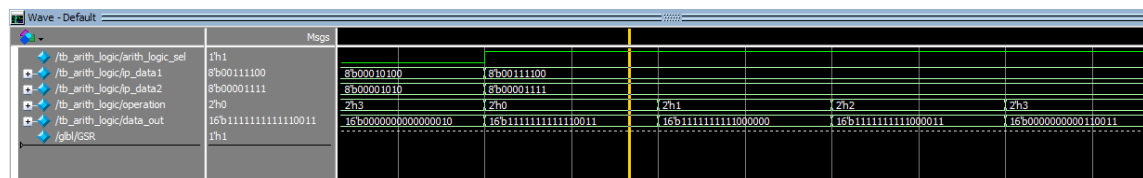


Figure 2-15: Simulation Results

The operation signal now decides between the logical operations, with operation = 0 being nand_op, operation = 1 being nor_op, and so on. Verify the data_out value as per the operation selected.

3-7. Exit the simulator.

3-7-1. Select **Simulate > End Simulation** to close the simulator.

3-7-2. Click **Yes** to confirm.

3-8. Close Questa Sim.

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

3-8-2. Click **Yes** to close Questa Sim.

Synthesizing the arith_logic Module

Step 4

In this step, you will synthesize the *arith_logic* SystemVerilog design to show that the packed, unpacked structures and packed unions are supported by Vivado Synthesis tool.

4-1. Explore the synthesis settings available.

4-1-1. Click **Settings** under the Flow Navigator and click **Synthesis**.

Note that the strategy is set to **Vivado Synthesis Defaults**.

4-1-2. Review the other synthesis options.

4-1-3. Click **OK** after reviewing the other options.

4-2. Run design synthesis based on the settings selected.

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

4-2-2. Select **View Reports** in the Synthesis Completed dialog box to view any of the reports generated after synthesis.

4-2-3. Click **Cancel** to close the Synthesis Completed dialog box.

4-3. Close the Vivado Design Suite.

4-3-1. Select **File > Exit**.

The Exit Vivado dialog box opens.



Figure 2-16: Exit Vivado Dialog Box

4-3-2. Click **OK**.

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

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

4-4. [Optional] [Only for local VMs—not for CloudShare] Clean up the file system.

Using the GUI:

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

4-4-2. Select `sv_struct`.

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

-- OR --

Using the command line:

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

4-4-5. Enter the following command to delete the contents of the workspace:

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

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

Summary

In this lab, you created a new Vivado Design Suite project and learned how to write a simple SystemVerilog module to create packed structures.

You created objects for these structures and accessed their members. You also used them to perform either arithmetic or logical operations.

You also saw the benefits of using them in the code, as they have the ability to store variables with different data types.

Answers

```

`timescale 1ns / 1ps

// User defined enumerated data type declaration
typedef enum {add=0, sub, mul, div} arith_operation;
typedef enum {nand_op=0, nor_op, not_op, xor_op} logic_operation;
// User defined Packed Structure declaration
typedef struct packed { arith_operation arithmetic_op;
                        logic_operation logical_op;
                        logic [7:0] data1;
                        byte data2;
                        logic [15:0] arith_result;
                        logic [15:0] logic_result;
                        } arith_logic_info;

module arith_logic (    input logic arith_logic_sel,                // Selects between arith or
logic operations
                        input int operation,                        // Selects which arith or
logic operations is to be done
                        input logic [7:0] ip_data1, ip_data2,      // Two data inputs
                        output logic [15:0] data_out);              // Output of the selected
operation
    always @ (*)
    begin
        // Creating two objects arith_data and logic_data
        arith_logic_info arith_data, logic_data;
        // If Arithmetic operation is selected
        if (arith_logic_sel==0)
        begin
            // Read the input data
            arith_data.arithmetic_op = arith_operation'(operation);
            arith_data.logical_op = logic_operation'(0);

```

```
        arith_data.data1 = ip_data1;
        arith_data.data2 = ip_data2;

    // Do the following arithmetic operations
    // add, sub, mul, div
    case (arith_data.arithmetic_op)           // Accessing enum data type
through arith_data input
    // As per the value of enum, do the arithmetic operations
    // Accesing the Packed Structure for two datas and storing the result in an
Packed Union

        add : arith_data.arith_result = arith_data.data1 + arith_data.data2;
        sub : arith_data.arith_result = arith_data.data1 - arith_data.data2;
        mul : arith_data.arith_result = arith_data.data1 * arith_data.data2;
        div : arith_data.arith_result = arith_data.data1 / arith_data.data2;

    endcase

    // Write the result to the output
    data_out = arith_data.arith_result;
end
else
    // If Logical operation is selected
    begin
    // Read the input data
    logic_data.arithmetic_op = arith_operation'(0);
    logic_data.logical_op = logic_operation'(operation);
    logic_data.data1 = ip_data1;
    logic_data.data2 = ip_data2;

    // Do the following logical operations
    // nand, nor, not, xor
    case (logic_data.logical_op)           // Accessing enum data type through
logic_data input
    // As per the value of enum, do the logic operations
```

```

// Accesing the Packed Structure for two datas and storing the result in an
Packed Union
    nand_op : logic_data.logic_result = ~((logic_data.data1) &
(logic_data.data2));
    nor_op  : logic_data.logic_result = ~((logic_data.data1) |
(logic_data.data2));
    not_op   : logic_data.logic_result = ~ (logic_data.data1);
    xor_op   : logic_data.logic_result = ((logic_data.data1) ^
(logic_data.data2));
endcase
// Write the result to the output
data_out = logic_data.logic_result;
end
end
endmodule
```


Lab 3: Unions

2021.1

Abstract

This lab introduces unions in SystemVerilog.

This lab should take approximately 30 minutes.

Objectives

After completing this lab, you will be able to:

- Write code for unions in SystemVerilog
- Simulate and synthesize a SystemVerilog design using the Vivado® Design Suite

Introduction

The arith_logic SystemVerilog model used in this lab works as a miniature 8-bit ALU. It performs four arithmetic or four logical operations on 8-bit operands.

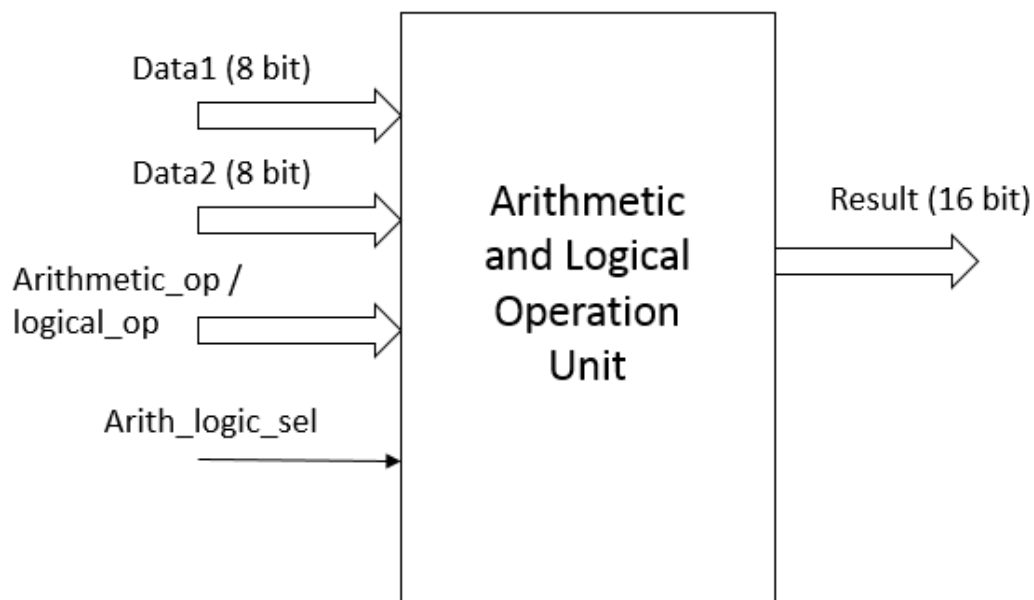


Figure 3-1: Block Diagram

The enumerated type variable `arithmetic_op` chooses between the four arithmetic operations that are available: addition, subtraction, multiplication, and division. Similarly, the `logical_op` variable decides between the four logical operations that are available: `nand`, `nor`, `not`, and `xor`.

These operations are done on two data signals, which are specified in the packed structure called `arith_logic_info`. The final result is of 16 bits.

The `arith_logic_sel` signal selects between the arithmetic and logical operations.

You will find the following signals in this lab:

- `arith_logic_sel` – Selects between arithmetic and logical operations.
- `operation` – Selects which arithmetic or logical operation is to be done.
- `ip_data1` and `ip_data2` – Two 8-bit data inputs.
- `data_out` – Output of the selected operation.

Apart from these, there are two objects created in this module, through which the structures and their members can be accessed:

- `arith_data` – Object of the packed structure `arith_logic_info` performing arithmetic operations.
- `logic_data` – Object of the packed structure `arith_logic_info` performing logical operations.

Understanding the Lab Environment

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

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

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

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

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

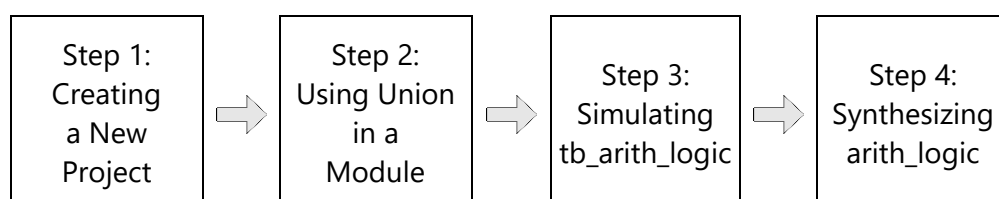
Environment Variable Name	Description
\$TRAINING_PATH	<p>Points to the space allocated for students to work through the labs. This directory includes prebuilt images and starting points for the labs and demos.</p> <p>The customer training VM sets \$TRAINING_PATH to the <code>/home/xilinx/training</code>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

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

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

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

General Flow



Creating a New Vivado Design Suite Project

Step 1

You will launch the Vivado Design Suite and create a new project.

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*.

"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).

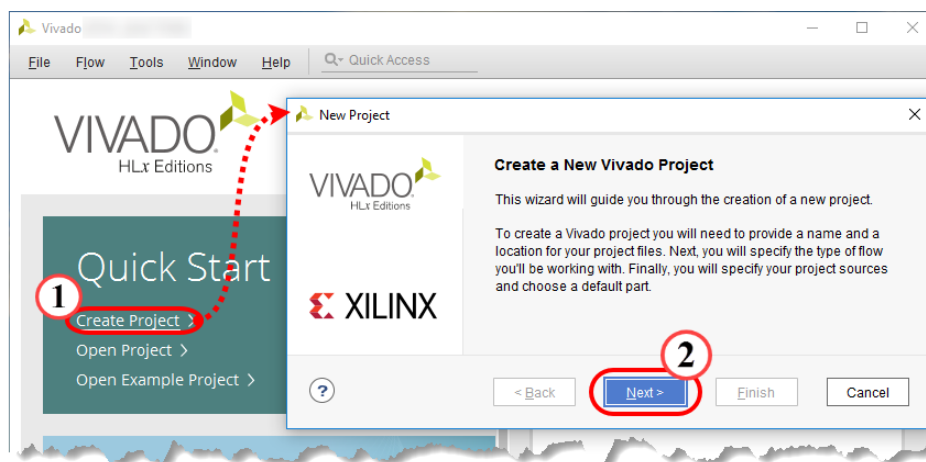


Figure 3-2: Creating a New Vivado Design Suite Project

This will launch the New Project Wizard.

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

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

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

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

\$TRAINING_PATH/sv_union/lab/KCU105

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

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

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

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

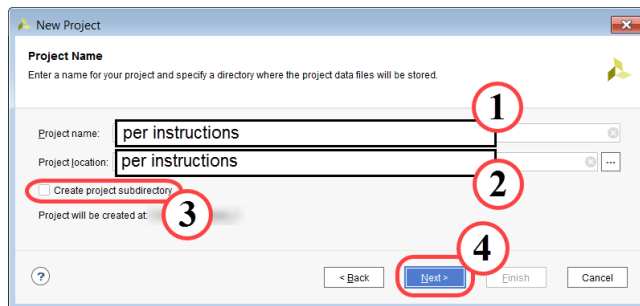


Figure 3-3: Entering the Project Name and Location

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

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

1-3-5. Select RTL Project.**1-3-6. Select Do not specify sources at this time**, which creates a blank project.

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

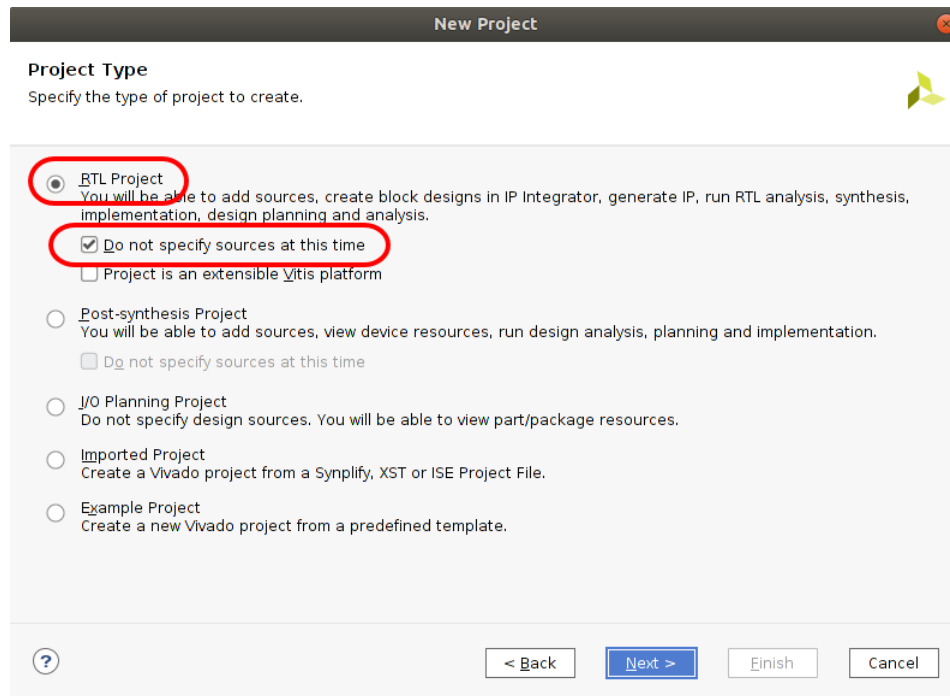


Figure 3-4: Specifying Project Options

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

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

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

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

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

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

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

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

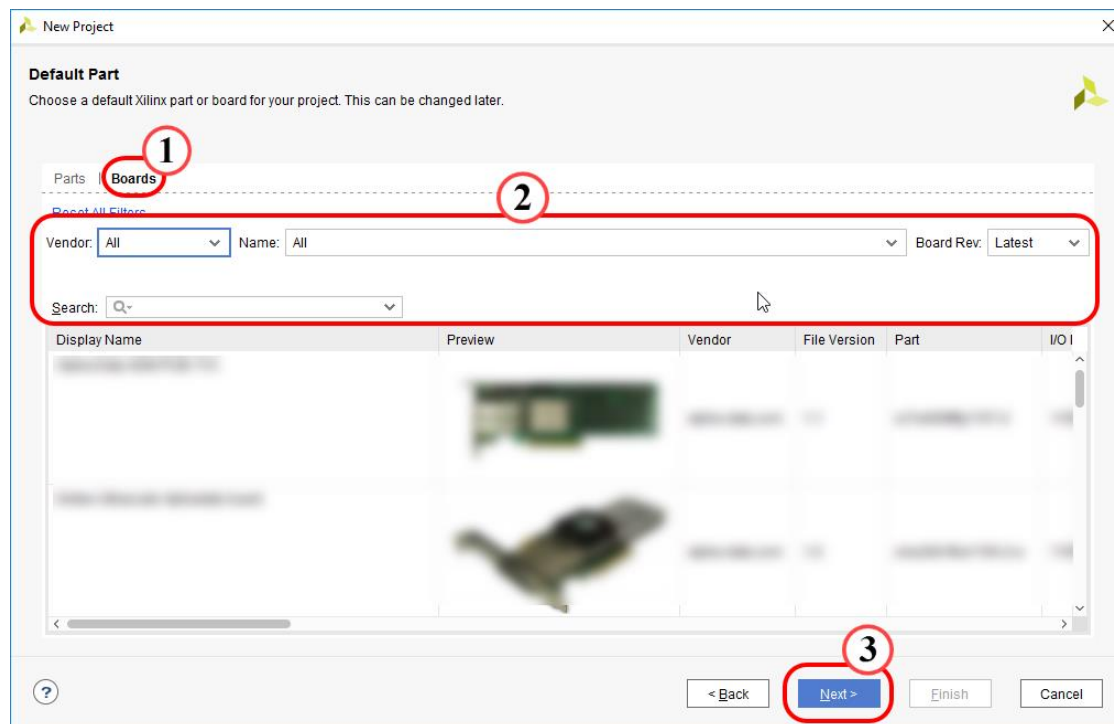


Figure 3-5: Selecting the Board for the Project

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

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

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

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

Using union in the Arithmetic and Logic Module

Step 2

In this step, you will edit a SystemVerilog module that will perform arithmetic or logical operations, depending upon the input given by the user.

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

2-1-1. Select **Add Sources** from the Flow Navigator tab, under Project Manager.

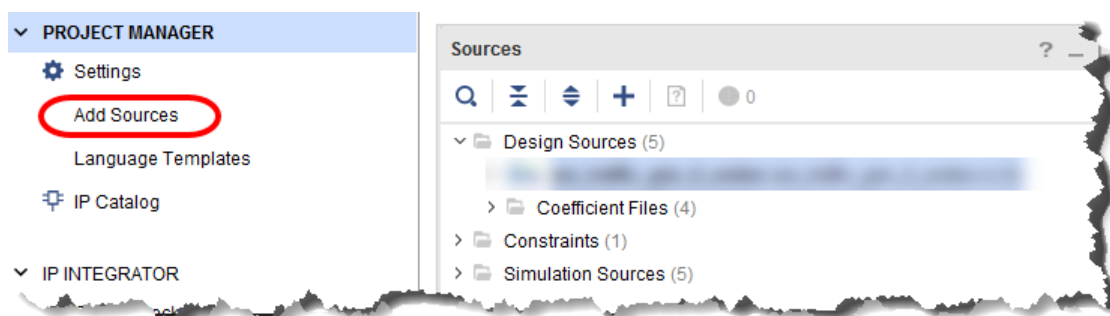


Figure 3-6: Selecting Add Sources

The Add Sources dialog box opens, allowing you to add HDL source files to the project.

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

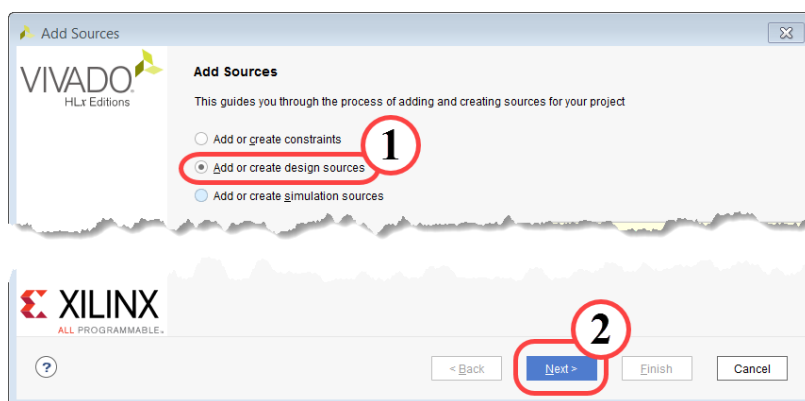


Figure 3-7: Selecting Add or Create Design Sources

2-1-3. Click **Next** to begin selecting source files.

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 `$sv_union/support/src`.

2-1-6. Select **arith_logic**.

2-1-7. Click **OK** in the Add Source Files dialog box to select the file(s).

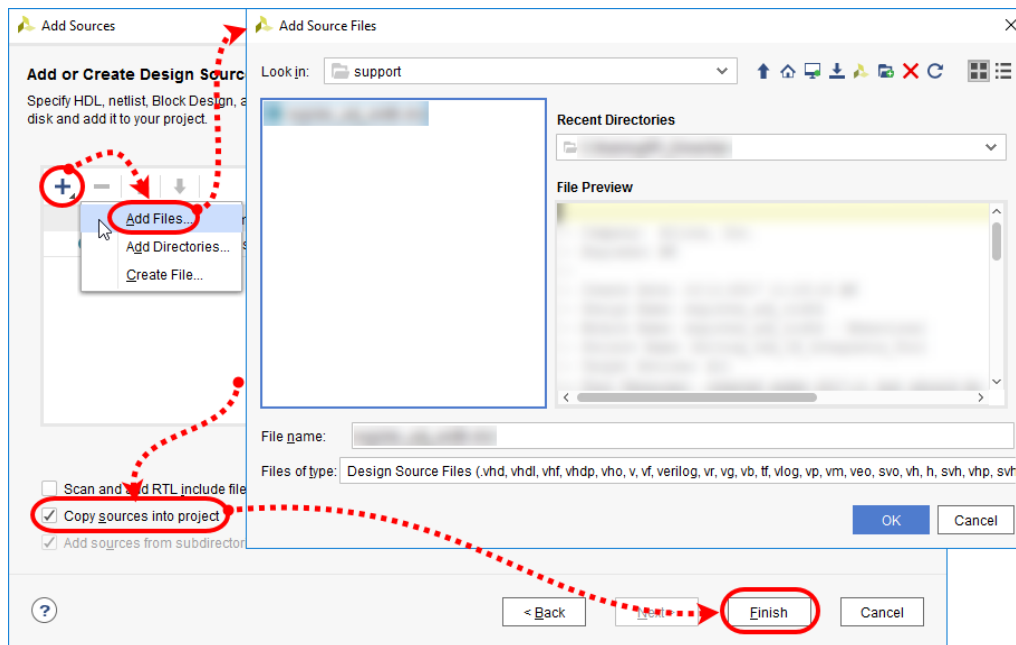


Figure 3-8: Selecting Add Files

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. Open and review an existing SystemVerilog module.

2-2-1. Expand **design sources** > **arith_logic** in the Sources window.

2-2-2. Double-click to open the file.

Observe that the enumerated type variables `arith_operation` and `logic_operation` are declared at the start. This module performs two kinds of operations: arithmetic or logical.

You can select between them with the help of the `arith_logic_sel` signal mentioned in the *if-else* statement.

The structure has six members: `arithmetic_op`, `logical_op`, `data1`, `data2`, `arith_result`, and `logic_result`. `data1` is of the logic data type and `data2` is of byte.

The other two members are of enum type, declared as `arith_operation`, `logic_operation` earlier.

2-2-3. Review the arithmetic and logical operations of the code section and the output assigned to the port.

2-3. Now modify the code by adding one more structure to define only the two outputs arithmetic and logic outputs using union.

2-3-1. Remove the arith_result and logic_result members of the struct arith_logic_info.

2-3-2. Create a struct named arith_logic_result near line no.34.

Inside this struct create a union with two members arith_result and logic_result of logic type of 16 bits.

2-3-3. Create an object for the new structure arith_logic_result named final_result.

2-3-4. Replace the code of storing the result in a packed union in arithmetic and logical operations of the code.

2-3-5. Finally, assign the output to data_out using the result stored in a packed union.

2-3-6. Select **File > Text Editor > Save File** to save the file.

Note: Refer to the Answers section at the end of this lab to verify the code.

2-4. Adding a new SystemVerilog testbench file called tb_arith_logic.

HDL simulation files can be added to the design at any time.

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

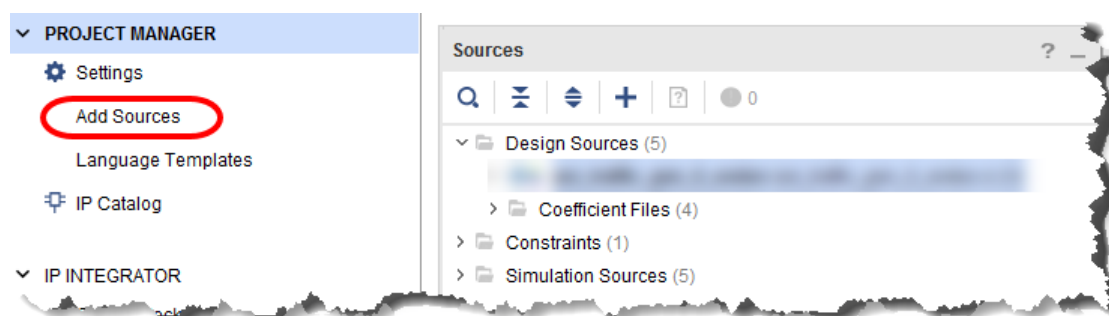
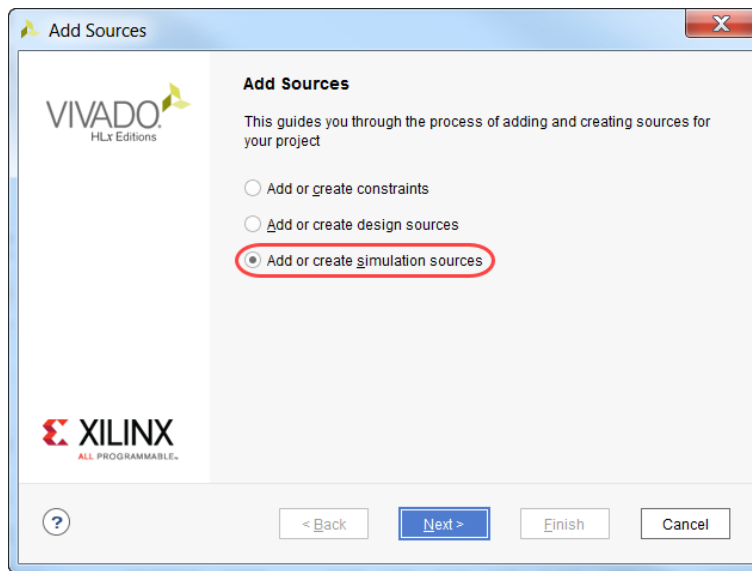


Figure 3-9: Selecting Add Sources

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

2-4-3. Click **Next.****Figure 3-10: Add Sources Dialog Box**

- 2-4-4.** Click the **Plus (+)** icon to open the pop-up menu.
- 2-4-5.** Select **Add Files** to open the Add Source Files dialog box which allows you to browse to the desired directory.
- 2-4-6.** Browse to the `$TRAINING_PATH/sv_union/support/src` directory if it is not open already.
- 2-4-7.** Select **tb_arith_logic.sv**.
- 2-4-8.** Click **OK** in the Add Source Files dialog box.
- 2-4-9.** Click **Finish** to add the file(s) to the project.
- 2-4-10.** Observe the SystemVerilog testbench file to see how the initialization is done.
- 2-4-11.** Observe how the stimulus is given to the module ports.

Simulating the tb_arith_logic Module

Step 3

In this step, you will simulate the tb_arith_logic module and observe the outputs.

The following instructions are for Vivado Simulator users only. Mentor Graphics Questa users can proceed to the instruction that begins with "The following instructions are for Questa users only".

3-1. Run the simulation.

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

The simulation runs for the default **1us**.

3-1-2. Use the zoom options in the toolbar to scale the waveform window.

3-2. Change the radix and verify the simulation result.

3-2-1. Verify that the arith_logic_sel signal decides between the arithmetic and logical operation in the waveform window.

3-2-2. Right-click the ip_data1, ip_data2, and data_out signals and select **Radix > Unsigned Decimal**.

This will make it easier to analyze the waveform while checking the arithmetic operations (arith_logic_sel = 0).

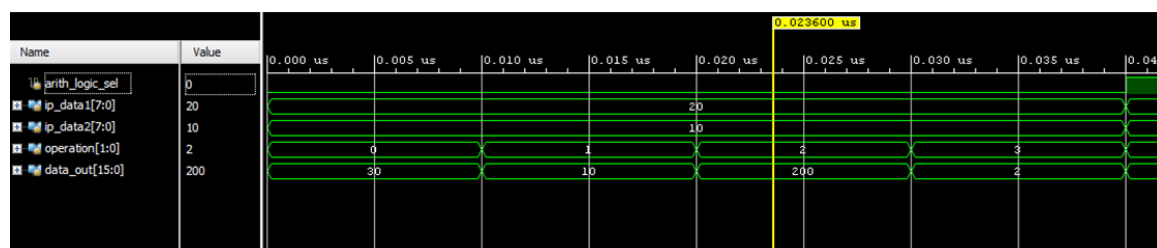


Figure 3-11: Simulation Result Showing Arithmetic Operation

The operation signal decides between the arithmetic operations, with operation = 0 being add, operation = 1 being subtract, and so on. Verify the data_out value as per the operation selected.

3-2-3. Right-click the ip_data1, ip_data2, and data_out signals again and select **Radix > Binary**.

This will make it easier to analyze the waveform while checking the logical operations (arith_logic_sel = 1).

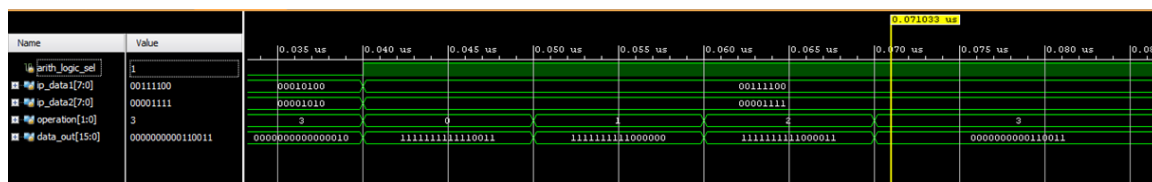


Figure 3-12: Simulation Result Showing Logical Operation

This time, the operation signal decides between the logical operations, with operation = 0 being nand_op, operation = 1 being nor_op, and so on. Verify the data_out value as per the operation selected.

3-3. Exit the simulator.

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

3-3-2. Click **OK** to confirm.

3-3-3. Click **Discard** to exit without saving the waveform.

The following instructions are for Questa users only. Vivado simulator users can proceed to the next step of this lab.

3-4. Set the simulation target to Questa Advanced Simulator.

3-4-1. Select **Settings** under Project Manager from the Flow Navigator.

3-4-2. Set the Target Simulator field to **Questa Advanced Simulator** under Simulation.

3-4-3. Click **Yes** in the Target Simulation dialog box.

This change the target simulator to Questa Advanced Simulator.

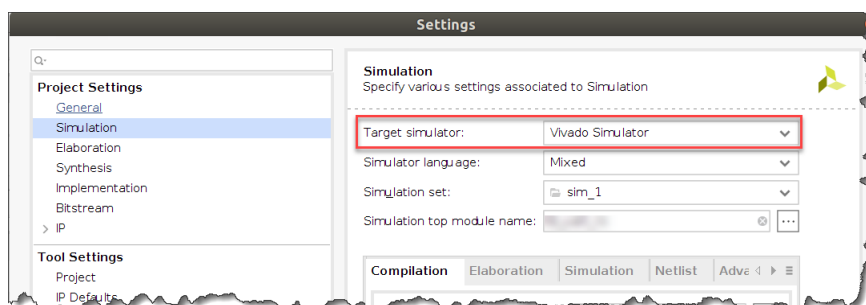


Figure 3-13: Simulation Settings

3-4-4. Click **OK** in the Settings dialog box.

3-5. Run the simulation.

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

3-5-2. Use the zoom options in the toolbar to scale the waveform window.

3-6. Change the radix and verify the simulation result.

3-6-1. Verify that arith_logic_sel signal decides between the arithmetic and logical operations in the waveform window.

3-6-2. Right-click the ip_data1, ip_data2, and data_out signals and select **Radix > Decimal**.

This will make it easier to analyze the waveform while checking arithmetic operations (arith_logic_sel = 0).

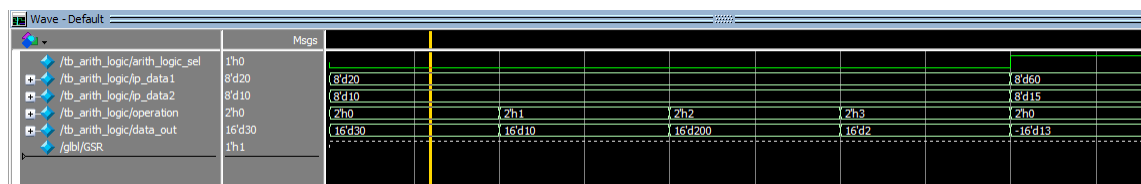


Figure 3-14: Simulation Results

The operation signal decides between the arithmetic operations, with operation = 0 being add, operation = 1 being subtract, and so on. Verify the data_out value as per the operation selected.

3-6-3. Right-click the ip_data1, ip_data2, and data_out signals again and select **Radix > Binary**.

This will make it easier to analyze the waveform while checking logical operations (arith_logic_sel = 1).

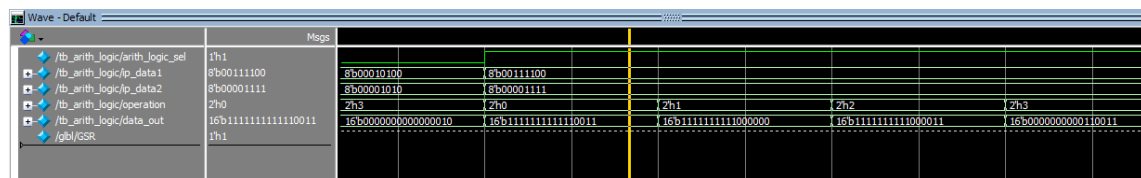


Figure 3-15: Simulation Results

The operation signal now decides between the logical operations, with operation = 0 being nand_op, operation = 1 being nor_op, and so on. Verify the data_out value as per the operation selected.

3-7. Exit the simulator.

3-7-1. Select **Simulate > End Simulation** to close the simulator.

3-7-2. Click **Yes** to confirm.

3-8. Close Questa Sim.

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

3-8-2. Click **Yes** to close Questa Sim.

Synthesizing the arith_logic Module

Step 4

In this step, you will synthesize the *arith_logic* SystemVerilog design to show that the packed, unpacked structures and packed unions are supported by Vivado Synthesis tool.

4-1. Explore the synthesis settings available.

4-1-1. Click **Settings** under the Flow Navigator and click **Synthesis**.

Note that the strategy is set to **Vivado Synthesis Defaults**.

4-1-2. Review the other synthesis options.

4-1-3. Click **OK** after reviewing the other options.

4-2. Run design synthesis based on the settings selected.

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

4-2-2. Select **View Reports** in the Synthesis Completed dialog box to view any of the reports generated after synthesis.

4-2-3. Click **Cancel** to close the Synthesis Completed dialog box.

4-3. Close the Vivado Design Suite.

4-3-1. Select **File > Exit**.

The Exit Vivado dialog box opens.



Figure 3-16: Exit Vivado Dialog Box

4-3-2. Click **OK**.

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

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

4-4. [Optional] [Only for local VMs—not for CloudShare] Clean up the file system.

Using the GUI:

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

4-4-2. Select **sv_union**.

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

-- OR --

Using the command line:

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

4-4-5. Enter the following command to delete the contents of the workspace:

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

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

Summary

In this lab, you created a new Vivado Design Suite project and learned how to write a simple SystemVerilog module to create packed structures.

You created objects for these structures and accessed their members. You also used them to perform either arithmetic or logical operations.

You also saw the benefits of using them in the code, as they have the ability to store variables with different data types.

Answers

```

`timescale 1ns / 1ps

// User defined enumerated data type declaration
typedef enum {add=0, sub, mul, div} arith_operation;
typedef enum {nand_op=0, nor_op, not_op, xor_op} logic_operation;
// User defined Packed Structure declaration
typedef struct packed { arith_operation arithmetic_op;
                        logic_operation logical_op;
                        logic [7:0] data1;
                        byte data2;
                        } arith_logic_info;
// User defined Unpacked Structure declaration
typedef      struct    {      // User defined Packed Union declaration
                        union packed {logic [15:0] arith_result;

                        logic [15:0] logic_result;
                        } result;
                        } arith_logic_result;
module arith_logic (      input logic arith_logic_sel,                // Selects between arith or
logic operations
                        input int operation,                        // Selects which arith or
logic operations is to be done
                        input logic [7:0] ip_data1, ip_data2,        // Two data inputs
                        output logic [15:0] data_out);                // Output of the selected
operation
    always @ (*)
    begin
        // Creating two objects arith_data and logic_data
        arith_logic_info arith_data, logic_data;
        arith_logic_result final_result;                // Creating an object for structure
arith_logic_result

```

```

// If Arithmetic operation is selected
if (arith_logic_sel==0)
begin
// Read the input data
    arith_data.arithmetic_op = arith_operation'(operation);
    arith_data.logical_op = logic_operation'(0);
        arith_data.data1 = ip_data1;
        arith_data.data2 = ip_data2;

// Do the following arithmetic operations
// add, sub, mul, div
    case (arith_data.arithmetic_op)           // Accessing enum data type
through arith_data input
        // As per the value of enum, do the arithmetic operations
        // Accesing the Packed Structure for two datas and storing the result in an
Packed Union
            add : final_result.result.arith_result = arith_data.data1 +
arith_data.data2;
            sub : final_result.result.arith_result = arith_data.data1 -
arith_data.data2;
            mul : final_result.result.arith_result = arith_data.data1 *
arith_data.data2;
            div : final_result.result.arith_result = arith_data.data1 /
arith_data.data2;
        endcase
        // Write the result to the output
        data_out = final_result.result.arith_result;
    end
else
// If Logical operation is selected
begin
// Read the input data
    logic_data.arithmetic_op = arith_operation'(0);

```

```
    logic_data.logical_op = logic_operation'(operation);
    logic_data.data1 = ip_data1;
    logic_data.data2 = ip_data2;

    // Do the following logical operations
    // nand, nor, not, xor
    case (logic_data.logical_op)           // Accessing enum data type through
logic_data input
    // As per the value of enum, do the logic operations
    // Accesing the Packed Structure for two datas and storing the result in an
Packed Union

        nand_op : final_result.result.logic_result = ~((logic_data.data1) &
(logic_data.data2));
        nor_op  : final_result.result.logic_result = ~((logic_data.data1) |
(logic_data.data2));
        not_op   : final_result.result.logic_result = ~ (logic_data.data1);
        xor_op   : final_result.result.logic_result = ((logic_data.data1) ^
(logic_data.data2));

    endcase
    // Write the result to the output
    data_out = final_result.result.logic_result;

end
end
endmodule
```


Lab 4: `always_ff` and `always_comb` Procedural Blocks

2021.1

Abstract

This lab will illustrate the usage of additional `always` blocks supported by SystemVerilog. You will edit the `security_sv` SystemVerilog module created in the previous lab to use `always_comb`, `always_latch` or `always_ff` blocks instead of `always@` procedural blocks.

This lab should take approximately 60 minutes.

Objectives

After completing this lab, you will be able to:

- Identify when to use `always_comb`, `always_latch`, and `always_ff` procedural blocks
- Explain the use of `priority` and `unique` keywords in `case` statements
- Illustrate how using `always_comb`, `always_latch`, and `always_ff` procedural blocks enables validation of designer intent

Introduction

To indicate the designer's intent, SystemVerilog adds three more procedural blocks:

- `always_comb` - combinational logic procedural blocks
- `always_latch` - latched logic procedural blocks
- `always_ff` - sequential logic procedural blocks

In this lab, you will edit the `security_sv` SystemVerilog module created in the previous lab and replace each of the four `always@` procedural blocks in the module with one of the `always_comb`, `always_latch`, or `always_ff` blocks. You will then synthesize the updated SystemVerilog module using the Vivado® synthesis tool and analyze the reports.

Based on the synthesis log report, you will include `priority` or `unique` keywords in the code to fix warning messages issued by the synthesis tool. Priority and unique keywords are intended to notify simulation and synthesis tools how to interpret case statements.

You will also conduct an experiment to see if the synthesis tool issues a warning when an incorrect `always` block is used.

Understanding the Lab Environment

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

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

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

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

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

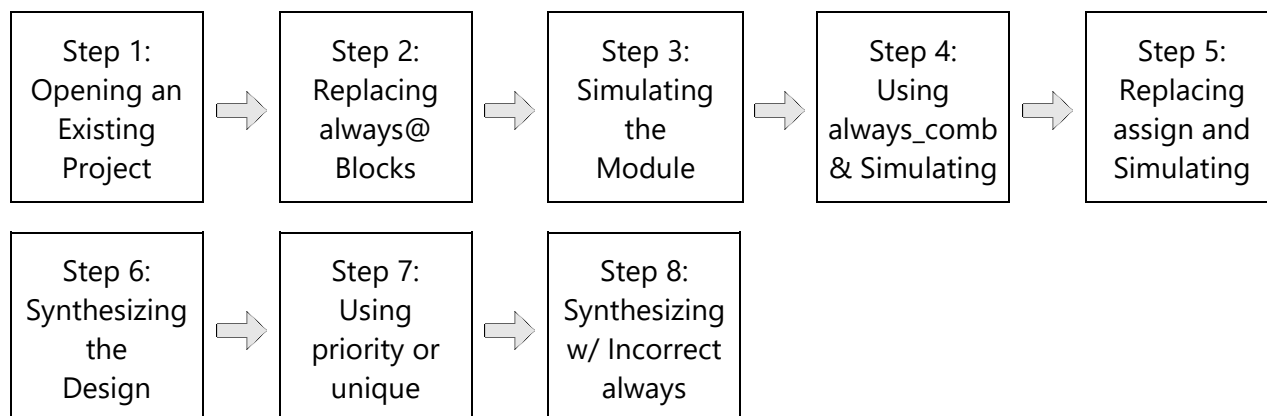
Environment Variable Name	Description
<code>\$TRAINING_PATH</code>	<p>Points to the space allocated for students to work through the labs. This directory includes prebuilt images and starting points for the labs and demos.</p> <p>The customer training VM sets <code>\$TRAINING_PATH</code> to the <code>/home/xilinx/training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

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

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

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

General Flow



Opening an Existing 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*.

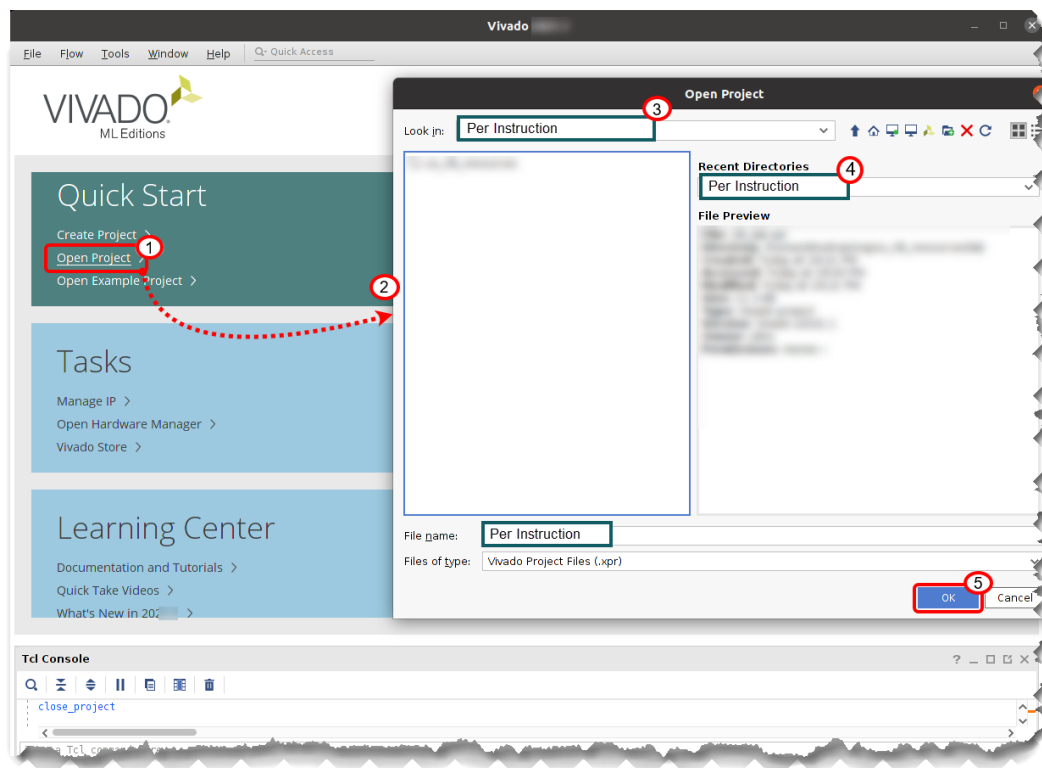
1-2. Open the existing Vivado Design Suite project *always_blocks.xpr*.

1-2-1. Click **Open Project** from the Quick Start section (1).

The Open Project dialog box opens (2).

1-2-2. Browse to the \$TRAINING_PATH/always_ff_always_comb/lab/KCU105 directory in the Look in field (3).

Note: The drop-down arrow shows the directory hierarchy.

1-2-3. Select `always_blocks.xpr` (4).**Figure 4-1: Opening an Existing Project****1-2-4. Click **OK** to open the selected project (5).**

The project now opens in the Vivado Design Suite.

Replacing `always@` Procedural Blocks

Step 2

You will edit the `security_sv` SystemVerilog module that was created in the previous lab to model a security system based on a finite state machine. The four `always@` procedural blocks in the module will be replaced with `always_comb`, `always_latch`, or `always_ff` blocks.

2-1. Open the SystemVerilog module and analyze the procedural blocks.

2-1-1. Expand **Design Sources** in the Sources window and double-click **`security_sv.sv`**.

The SystemVerilog file opens.

2-1-2. Observe that the following four procedural blocks are used in the module:

- The first procedural block increments the state machine every clock cycle
- The second procedural block determines what the next state should be
- The third procedural block generates the output of the state machine
- The fourth procedural block is used to count for 100 clock cycles before sounding the alarm.

2-2. Replace the four `always@` blocks with `always_comb`, `always_latch`, or `always_ff` blocks.

2-2-1. Determine which one of the `always_comb`, `always_latch`, or `always_ff` blocks is suitable for each of the four blocks.

Hint: The sensitivity list of an `always@` block provides a good indication of the logic being modeled by that block. If the block is edge sensitive (e.g., `always@(posedge clk)`), then `always_ff` is suitable. If the block is level sensitive (e.g., `always@(rst)`), then either `always_comb` or `always_latch` may be suitable. `always_latch` is used when the code does not cover all possible outcomes.

If the design intent of an `always` procedure is to generate combinatorial signals, `always_comb` can be used.

When the design intent is to generate latched signals (implied through selective non-assignment/retention under some conditions), an `always_latch` procedure can be used.

This helps the tools to distinguish between intended and unintended latches in the design. Unintended latches result from incompletely specified (combinatorial) procedural blocks and synthesis tools issue warnings.

2-2-2. Replace all four `always@` blocks with the appropriate procedural block.

Recall that `always_comb` and `always_latch` infer sensitivity to signals and do not require the specification of a sensitivity list.

`always_ff` requires the specification of a sensitivity list that infers edge-sensitive flip-flops.

2-2-3. Refer to the Answers section of this lab to verify that your choices are correct.

2-2-4. Select **File > Text Editor > Save File** to save the file.

Simulating the Module

Step 3

Here you will simulate the updated SystemVerilog module to check the functionality of the different always blocks.

The following instructions are for Vivado simulator users only. Mentor Graphics Questa users can proceed to the instruction that begins with "The following instructions are for Questa users only".

3-1. Simulate the design by using the provided testbench.

3-1-1. Review the contents of the testbench (*security_sv_tb*) if you did not do so in the previous lab to understand how it verifies the UUT, *security_sv*.

The testbench is located under Simulation Sources > sim_1.

3-1-2. Select **Simulation > Run Simulation > Run Behavioral Simulation** from the Flow Navigator.

The simulation runs for the default **1us**.

3-2. Add signals whose waveforms need to be analyzed.

3-2-1. Select **security_sv_tb > uut** from the Scopes pane.

You can now view the UUT instance that is the lower-level module (*security_sv*) relative to the testbench.

The Simulation Objects pane now shows the signals at the UUT level of hierarchy.

3-2-2. Drag-and-drop signals such as *curr_state*, *next_state*, *sensors*, *start_count*, *count_done*, or any other signal of interest to the waveform window.

3-3. Change the radix of the signals as necessary.

3-3-1. Right-click the signal keypad in the waveform window and select **Radix > Binary**.

This will make it easier to analyze the waveform.

3-3-2. Change the radix of any other signal of interest in the same way.

3-4. Restart and rerun the simulation in interactive mode.

3-4-1. Select **Run > Restart** to restart the simulation.

You can also click the Restart icon from the toolbar.

3-4-2. Select **Run > Run For**.

3-4-3. Enter **50us** as the time in the Run For dialog box.

3-4-4. Click **OK** to start the simulation.

This will fully validate the transitions described in the testbench.

3-5. Verify the simulation result.

3-5-1. Use the zoom options in the toolbar to scale the waveform window.

3-5-2. Observe the waveforms corresponding to signals such as *clk*, *keypad*, *sensors*, *curr_state*, *start_count*, etc., to verify that the functionality of the FSM is still correct.

Specifically, observe how the outputs (*is_armed*, *is_wait_delay*, and *alarm_siren*) transition in sync with the variable *curr_state*.

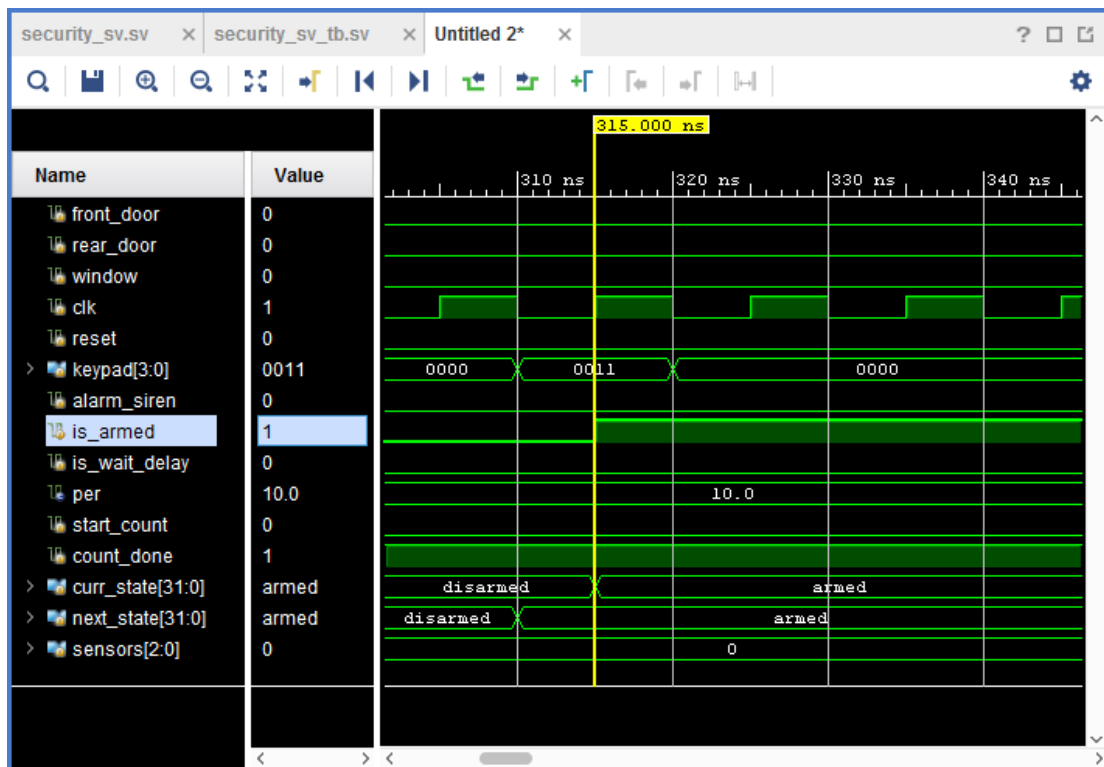


Figure 4-2: Simulation Results

Notice that unlike *next_state*, *curr_state* gets updated only at the rising clock edge, making the outputs synchronous with the clock.

The simulation will end once all tests have been cleared. You should see "test passed" displayed in the Tcl Console window.

3-5-3. Select **File > Close Simulation**.

3-5-4. Click **OK** to confirm.

3-5-5. Click **Save** to save changes to the waveform configuration.

3-5-6. Click **Save**.

3-5-7. Click **Yes** to add the waveform configuration to the project.

The following instructions are for Questa users only. Vivado simulator users can proceed to the next step of this lab.

3-6. Set the simulation target to Questa Advanced Simulator.

3-6-1. Select **Settings** under Flow Navigator.

3-6-2. Set the Target Simulator field to **Questa Advanced Simulator** under Simulation.

3-6-3. Click **Yes** in the Target Simulation dialog box.

This changes the target simulator to Questa Advanced Simulator.

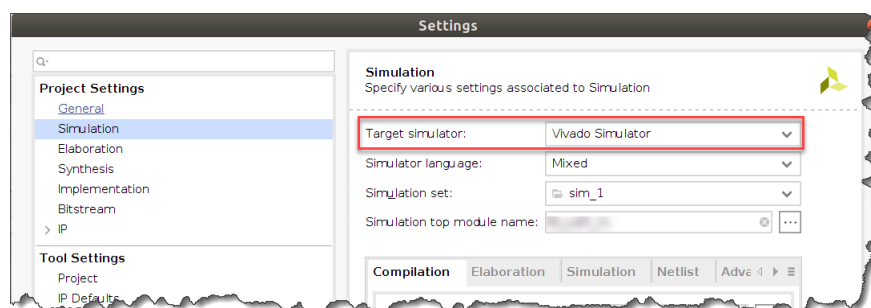


Figure 4-3: Simulation Settings

3-6-4. Click **OK** in the Settings dialog box.

3-7. Run the simulation.

3-7-1. Select **Simulation > Run Simulation > Run Behavioral Simulation** from the Flow Navigator.

3-8. Add signals whose waveforms need to be analyzed.

3-8-1. Select **View > Objects** if the Objects window is not opened.

3-8-2. Select **security_sv_tb > uut** in the sim-Default window.

All the top-level signals of the unit under test (*security_sv*) will be displayed in the Objects window.

3-8-3. Select signals of interest in the Object window.

3-8-4. Right-click the selected signals and select **Add Wave**.

The selected signals will be added to the default waveform window.

3-9. Change the radix of the signals as necessary.

3-9-1. Right-click the signal **keypad** in the waveform window and select **Radix > Binary**.

This will make it easier to analyze the waveform.

3-9-2. Change the radix of *delay_val* and *start_count* to **Decimal** in the same way.

3-10. Run the simulation for 50 us.

3-10-1. Select **Simulate > Restart**

3-10-2. Click **OK** in the Restart window.

3-10-3. Type **run 50us** in the Transcript window and press **<Enter>**.

3-10-4. Click **No** when asked if you want the simulation to finish.

The "test passed" comment should be displayed in the Transcript window, along with other comments corresponding to each state.

3-11. Verify the simulation result.

3-11-1. Select the **Wave** tab to view the waveforms.

3-11-2. Use the zoom options in the toolbar to scale the waveform window.

3-11-3. Observe the waveforms corresponding to signals such as *clk*, *keypad*, *sensors*, *curr_state*, *start_count*, etc. to verify that the functionality of the FSM is still correct.

Specifically, observe how the outputs (*is_armed*, *is_wait_delay*, and *alarm_siren*) transition in sync with the variable *curr_state*.

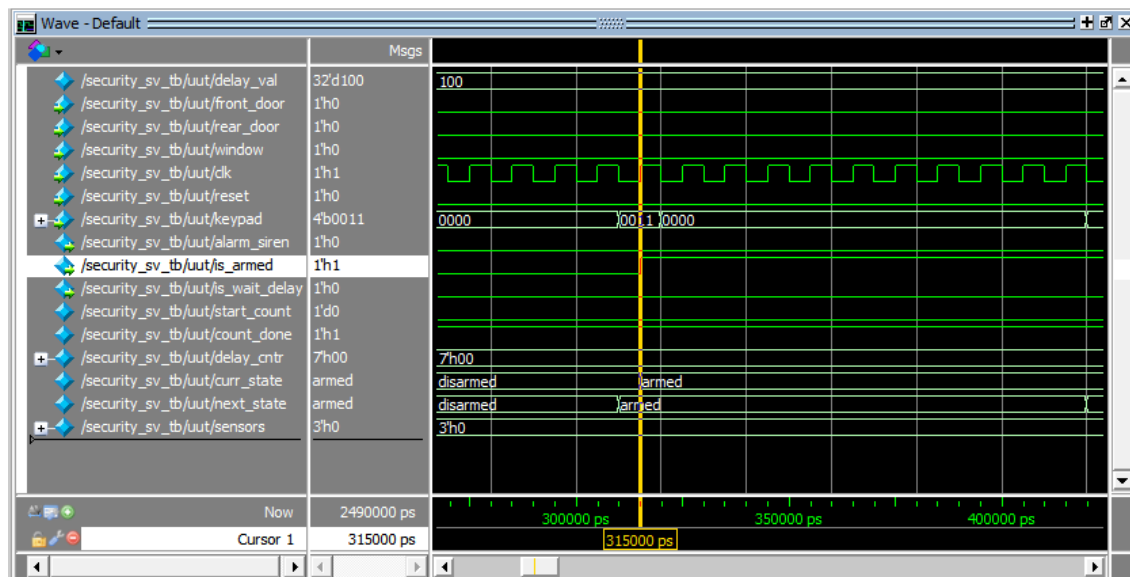


Figure 4-4: Simulation Results Showing Outputs Being in Sync with *curr_state*

Notice that unlike *next_state*, *curr_state* gets updated only at the rising clock edge, making the outputs synchronous with the clock.

3-12. Exit the simulator.

3-12-1. Select **Simulate** > **End Simulation** to close the simulator.

3-12-2. Click **Yes** to confirm.

3-13. Close Questa Sim.

3-13-1. Select **File** > **Quit**.

3-13-2. Click **Yes** to close Questa Sim.

Using `always_comb` to Generate the Outputs and Simulating Step 4

You will use the `always_comb` procedural block to generate the outputs of the state machine and observe how it impacts the transition of the outputs.

4-1. Replace the `always_ff` block used to generate the outputs with `always_comb`.

4-1-1. Identify the procedural block in the `security_sv` module that generates the state machine output values.

4-1-2. Replace the procedural block with `always_comb`.

Recall that `always_comb` infers sensitivity to signals and does not require the specification of a sensitivity list.

The updated code should look as shown in the figure below.

```
// procedural block to generate the state machine output values
// always_ff @ ( posedge clk ) begin
always_comb begin
    if (reset) begin
        is_armed      <= 1'b0 ;
        is_wait_delay <= 1'b0 ;
        alarm_siren   <= 1'b0 ;
    end
    else
    begin
        is_armed      <= ( next_state == armed );
        is_wait_delay <= ( next_state == wait_delay );
        alarm_siren   <= ( next_state == alarm );
    end
end
```

Figure 4-5: Using `always_comb` to Generate Outputs

4-1-3. Select **File** > **Text Editor** > **Save File** to save the file.

The following instructions are for Vivado simulator users only. Mentor Graphics Questa users can proceed to the instruction that begins with "The following instructions are for Questa users only".

4-2. Open the simulation settings.

4-2-1. Select **Settings** in the Flow Navigator, under Project Manager and go to **Simulation**.

The simulation settings opens and contains five sub-tabs: Compilation, Elaboration, Simulation, Netlist, and Advanced.

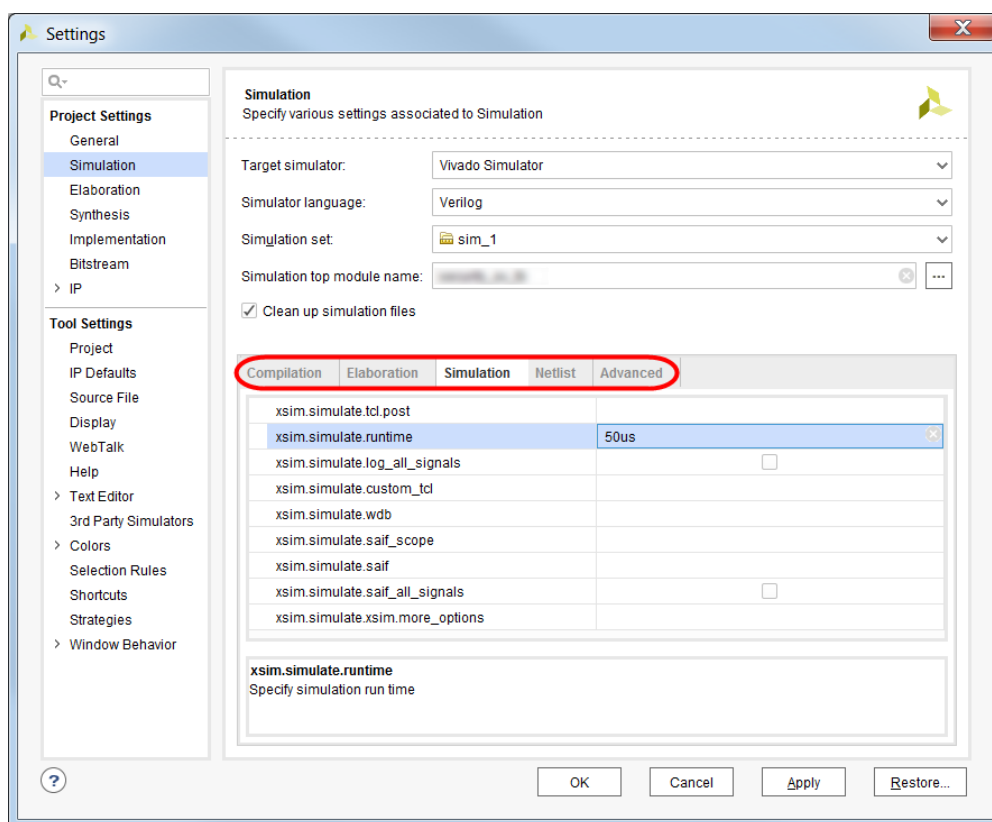


Figure 4-6: Vivado Simulator Settings

4-2-2. Select the **Simulation** tab.

4-2-3. Enter **50us** in the **xsim.simulate.runtime** field.

This would set the default simulation time to **50us**.

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

4-3. Run the simulation.

4-3-1. Select **Simulation** > **Run Simulation** > **Run Behavioral Simulation** from the Flow Navigator.

4-4. Verify the simulation result.

- 4-4-1. Select the **security_sv_tb_behav.wcfg** tab to view the waveforms.
- 4-4-2. Use the zoom options in the toolbar to scale the waveform window.
- 4-4-3. Observe the waveforms corresponding to signals such as *clk*, *keypad*, *sensors*, *curr_state*, *start_count*, etc., to verify that the functionality of the FSM is still correct.

Specifically, observe how the outputs (*is_armed*, *is_wait_delay*, and *alarm_siren*) transition.

When *always_comb* is used, the outputs no longer transition in sync with the synchronous *curr_state*. The outputs are now asynchronous and transition at the same time as *next_state*.

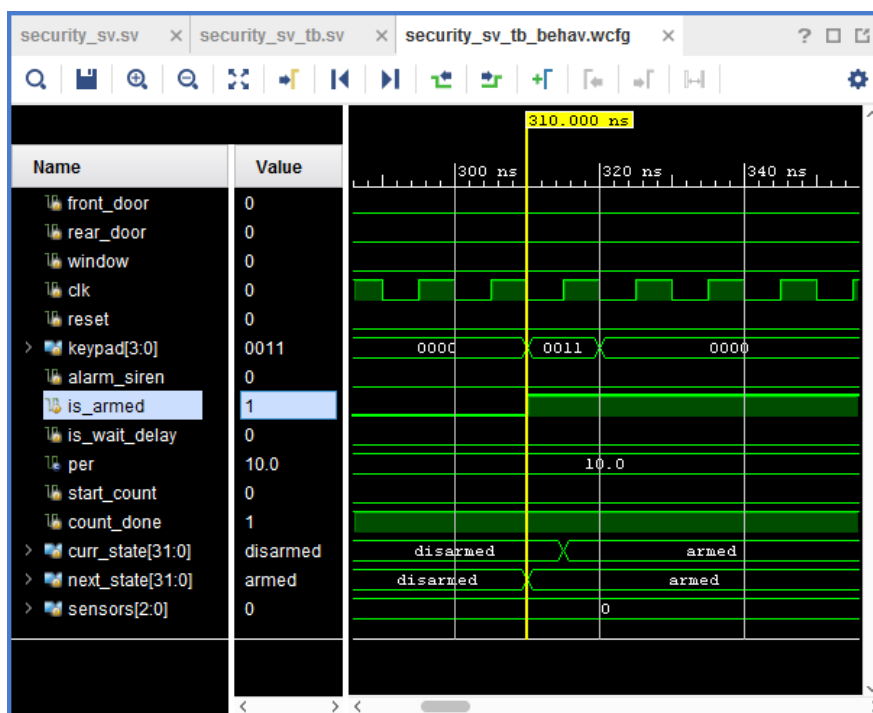


Figure 4-7: Simulation Results Showing Outputs Transitioning with next_state When always_comb is Used

This validates the fact that when an *always_comb* block is used, the tool will infer that the designer's intent is to model a combinational logic.

- 4-4-4. Select **File > Close Simulation**.
- 4-4-5. Click **OK** to confirm.

4-5. Replace *always_comb* with *always_ff*.

- 4-5-1. Replace the *always_comb* with *always_ff* as it originally was in order to keep the outputs synchronous.
- 4-5-2. Select **File > Text Editor > Save File** to save the file.

The following instructions are for Questa users only. Vivado simulator users can proceed to the next step of this lab.

4-6. Run the simulation.

- 4-6-1.** From the Flow Navigator select **Simulation** > **Run Simulation** > **Run Behavioral Simulation**.

4-7. Add signals whose waveforms need to be analyzed.

- 4-7-1.** Select **View** > **Objects** if the Objects window is not opened.

- 4-7-2.** Select **security_sv_tb** > **uut** in the sim-Default window.

All the top-level signals of the unit under test (*security_sv*) will be displayed in the Objects window.

- 4-7-3.** Select signals of interest in the Object window.

- 4-7-4.** Right-click the selected signals and select **Add Wave**.

The selected signals will be added to the default waveform window.

4-8. Change the radix of the signals as necessary.

- 4-8-1.** Right-click the signal **keypad** in the waveform window and select **Radix** > **Binary**.

This will make it easier to analyze the waveform.

- 4-8-2.** Change the radix of *delay_val* and *start_count* to **Decimal** in the same way.

4-9. Restart the simulation.

- 4-9-1.** Select **Simulate** > **Restart**.

- 4-9-2.** Click **OK** in the Restart dialog box.

4-10. Run the simulation for 50 us.

- 4-10-1.** Type **run 50us** in the Transcript window and press **<Enter>**.

- 4-10-2.** Click **No** when asked if you want the simulation to finish.

4-11. Verify the simulation result.

- 4-11-1.** Select the **Wave** tab to view the waveforms.

- 4-11-2.** Use the zoom options in the toolbar to scale the waveform window.

- 4-11-3.** Observe the waveforms corresponding to signals such as *clk*, *keypad*, *sensors*, *curr_state*, *start_count*, etc. to verify that the functionality of the FSM is still correct.

Specifically, observe how the outputs (*is_armed*, *is_wait_delay*, and *alarm_siren*) transition.

When *always_comb* is used, the outputs no longer transition in sync with the synchronous *curr_state*. The outputs are now asynchronous and transition at the same time as *next_state*.

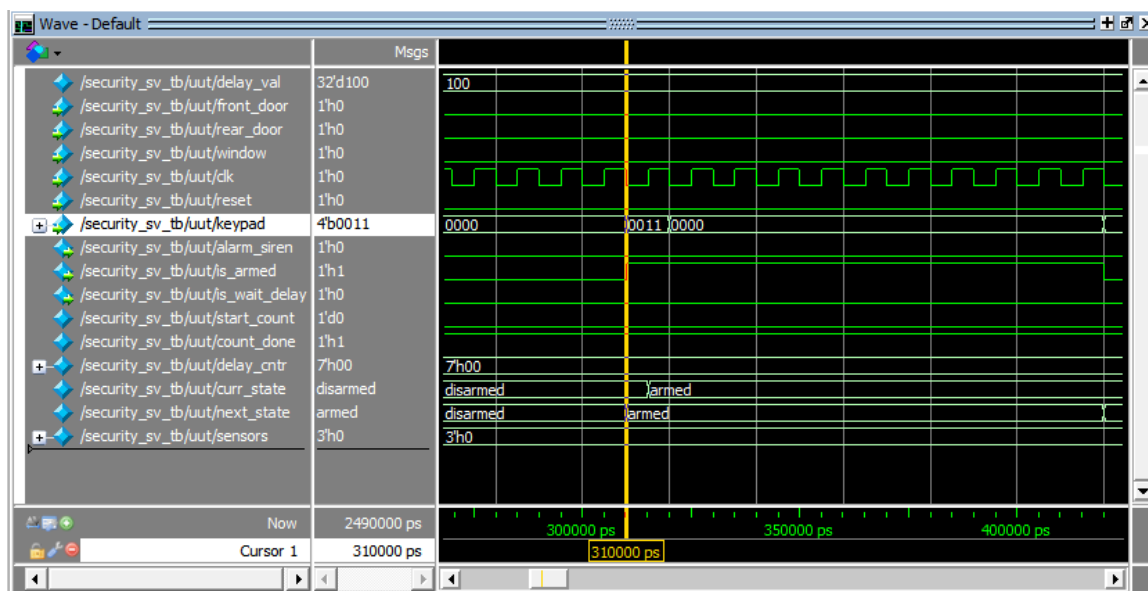


Figure 4-8: Simulation Results Showing Outputs Transitioning with next_state When always_comb Is Used

This validates the fact that when an *always_comb* block is used, the tool will infer that the designer's intent is to model a combinational logic.

4-12. Close Questa Sim.

4-12-1. Select **File > Quit**.

4-12-2. Click **Yes** to close Questa Sim.

4-13. Replace *always_comb* with *always_ff*.

4-13-1. Replace the *always_comb* with *always_ff* as it originally was in order to keep the outputs synchronous.

4-13-2. Select **File > Save** to save the file.

Replacing assign Statements and Simulating

Step 5

You will replace the three *assign* statements in the module with *always_comb*, *always_latch*, or *always_ff* blocks.

5-1. Open the SystemVerilog module and analyze the assign statements.

5-1-1. Select the **security_sv.sv** tab to view the file.

5-1-2. Observe that the following three *assign* statements are used in the module:

- The first *assign* statement is used to update the value of *sensors*.
- The second *assign* statement is used to update the value of *start_count*.
- The third *assign* statement is used to update the value of *count_done*.

5-2. Replace the assign statements with the appropriate always block.

5-2-1. Determine which one of the *always_comb*, *always_latch*, and *always_ff* blocks is suitable for replacing the *assign* statements.

5-2-2. Replace the three *assign* statements with the appropriate procedural block.

Recall that *always_comb* and *always_latch* do not require the specification of a sensitivity list.

5-2-3. Refer to the Answers section of this lab to verify if your choices are correct.

5-2-4. Select **File > Text Editor > Save File** to save the file.

The following instructions are for Vivado simulator users only. Mentor Graphics Questa users can proceed to the instruction that begins with "The following instructions are for Questa users only".

5-3. Run the simulation.

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

5-3-2. Click **Yes** to close the simulation and re-launch.

5-4. Verify the simulation result.

5-4-1. Select the waveform tab.

5-4-2. Observe the waveforms corresponding to signals such as *clk*, *keypad*, *sensors*, *curr_state*, *start_count*, etc., to verify that the functionality of the FSM is correct.

5-5. Exit the simulator.

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

5-5-2. Click **OK** to confirm.

5-5-3. Click **Discard** if prompted to save changes to the waveform.

The following instructions are for Questa users only. Vivado simulator users can proceed to the next step of this lab.

5-6. Run the simulation.

5-6-1. Select **Simulation > Run Simulation > Run Behavioral Simulation** from the Flow Navigator.

5-7. Add signals whose waveforms need to be analyzed.

5-7-1. Select signals of interest in the Object window.

5-7-2. Right-click the selected signals and select **Add Wave**.

The selected signals will be added to the default waveform window.

5-8. Run the simulation for 50 us.

5-8-1. Select **Simulate > Restart**.

Click **OK** in the Restart dialog box.

5-8-2. Type **50us** in the Transcript window and press **<Enter>**.

5-8-3. Click **No** when asked if you want the simulation to finish.

5-9. Verify the simulation result.

5-9-1. Select the **Wave** tab to view the waveforms.

5-9-2. Use the zoom options in the toolbar to scale the waveform window.

5-9-3. Observe the waveforms corresponding to signals such as *clk*, *keypad*, *sensors*, *curr_state*, *start_count*, etc. to verify that the functionality of the FSM is still correct.

5-10. Exit the simulator.

5-10-1. Select **Simulate > End Simulation** to close the simulator.

5-10-2. Click **Yes** to confirm.

5-10-3. Select **File > Quit**.

5-10-4. Click **Yes** to close QuestaSim.

Synthesizing the Design in the Vivado Design Suite

Step 6

You will synthesize the *security_sv* module in the Vivado Design Suite and analyze the Synthesis Report.

6-1. Run synthesis.

6-1-1. Under the Flow Navigator, click **Run Synthesis**.

After the synthesis completes, the Synthesis Completed dialog box opens.

6-1-2. Select **View Reports**.

6-1-3. Click **OK**.

6-2. Analyze the Vivado synthesis report.

6-2-1. From the Reports window, double-click **Vivado Synthesis Report**.

6-2-2. Analyze the messages issued by the Vivado synthesis tool.

6-2-3. Open the Synthesis report and confirm that there are no latches inferred.

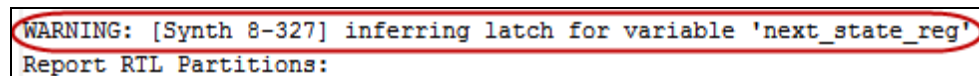
You can search for the keyword *latch* and you should not find any.

6-2-4. For experimentation, modify the enum declaration in *security_sv.sv* as follows to imply a 32-bit representation for *curr_state* and *next_state*:

```
enum {disarmed, armed, wait_delay, alarm} curr_state, next_state;
```

6-2-5. Re-synthesize the design and observe the warning message regarding latch inference in the report, in spite of *always_comb*.

Note that that *always comb* is only a design intent. Synthesis will achieve only if the code inference leads to the same behavior.



WARNING: [Synth 8-327] inferring latch for variable 'next_state_reg'
Report RTL Partitions:

Figure 4-9: Warning Message on Latch Inference

6-2-6. Observe the code in *security_sv* corresponding to the line number reported in the warning message.

Note that the warning pertains to the *case* statement in the procedural block that determines the next state. Since only 4 of the possible 2^{32} values have been covered in the case statement, the inference is to retain the previous values (for *next_state*) for the unspecified values.

Using the priority or unique Keyword to Avoid Latch Inference Step 7

You will update the *security_sv* module to include the *priority* or *unique* keyword to resolve the warning issued in the Synthesis Report.

7-1. Use the *priority* or *unique* keyword to resolve the warning.

7-1-1. Recall the use of *priority*, *unique*, and *unique0* keywords in *case*, *if*, and *if-else* statements.

Because *unique0* is not synthesizable, you should choose either *priority* or *unique* in the *case* statement.

7-1-2. Determine which of the two keywords is suited for this case.

Hint: Recall that the *priority* keyword implies that more than one match can be present and hence the priority of the case selection items is important. The *unique* keyword implies that only one match can be present and hence the order of the case selection items is irrelevant.

7-1-3. Update the *case* statement accordingly.

7-1-4. Save the **security_sv.sv** file.

7-2. Run synthesis.

7-2-1. Under the Flow Navigator, click **Run Synthesis**.

After the synthesis completes, the Synthesis Completed dialog box opens.

7-2-2. Select **View Reports**.

7-2-3. Click **OK**.

Notice that the warning message is no longer issued.

The following message is shown:

INFO: [Synth 8-294] found qualifier unique on case statement: implementing as parallel_case

Note: Good coding practice is to specify the right width of the reg/net and also have a default branch in the case statement. Unique/priority should be used after these if required.

Synthesizing with an Incorrect `always` Block

Step 8

Recall that the advantages of using `always_comb`, `always_latch`, and `always_ff` are:

- They make coding easier (`always_comb` and `always_latch` do not require sensitivity lists).
- They validate the intention of the designer (i.e., the synthesis tool warns if `always_latch` is used and no latch is inferred).
- They avoid gate-RTL mismatch which can occur if the sensitivity list is coded incorrectly.

In this step, you will validate the second point by intentionally using an inappropriate `always` block in the `security_v` module. You will synthesize the design and observe the warning issued by the synthesis tool.

8-1. Update `security_sv.sv` with an incorrect `always` block.

8-1-1. In the `security_sv.sv` file, replace the last procedural block that uses `always_ff` to count 100 clock cycles with `always_comb`.

8-1-2. Select **File** > **Text Editor** > **Save File** to save the file.

8-2. Run synthesis.

8-2-1. In the Flow Navigator, click **Run Synthesis**.

After the synthesis completes, the Synthesis Completed dialog box opens.

8-2-2. Select **View Reports**.

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

8-3. Analyze the Vivado synthesis report.

8-3-1. Double-click **Vivado Synthesis Report** in the Reports window.

8-3-2. Look for warning messages regarding the use of `always_comb`.

The following warning messages will be issued by the Vivado synthesis tool.

```
WARNING: [Synth 8-87] always_comb on 'delay_cntr_reg' did not result in combinational logic
```

Figure 4-10: Warning Messages When Incorrect `always` Blocks Are Used

8-3-3. Replace the incorrect `always_comb` used with the correct `always_ff`.

8-3-4. Save and close the file.

8-4. Close the Vivado Design Suite.

8-4-1. Select **File > Exit** to close the Vivado Design Suite.

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

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

8-5. [Optional] [Only for local VMs—not for CloudShare] Clean up the file system.

Using the GUI:

8-5-1. Using the graphical browser (Windows: press the **<Windows>** key + **<E>**; Linux: press **<Ctrl + N>**), navigate to `$TRAINING_PATH/always_ff_always_comb`.

8-5-2. Select **always_ff_always_comb**.

8-5-3. Press **<Delete>**.

-- OR --

Using the command line:

8-5-4. Open a terminal window (Windows: press the **<Windows>** key + **<R>**, then enter **cmd**; Linux: press **<Ctrl + Alt + T>**).

8-5-5. Enter the following command to delete the contents of the workspace:

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

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

Summary

Here you learned how to choose the correct always block for replacing the generic *always@* block and *assign* statements. You learned the advantages of using the always blocks over the generic *always@* procedural block. Also, you learned when to use the *priority* and *unique* keywords.

Answers

```
`timescale 1ns / 1ps

module security_sv(
    input front_door,
    input rear_door,
    input window,
    input clk,
    input reset,
    input [3:0] keypad,
    output logic alarm_siren, // replace all reg and wire with logic
    output logic is_armed,
    output logic is_wait_delay
);

// set the delay value (the number of clocks between a faulted zone and the
// alarm going off)
parameter delay_val = 100;

// Variables used for counting 100 (delay_val) clock cycles
logic start_count;
logic count_done;
logic [6:0] delay_cntr = 0 ; // Max value of delay_cntr is delay_val (i.e., d'100 or b'1100100)

// Enumerated types provide a means for defining a variable that has a
// restricted set of legal values. The values are represented with labels
// instead of digital logic values.
enum logic[1:0] {disarmed, armed, wait_delay, alarm} curr_state, next_state;

logic [2:0] sensors ; // used to combine inputs
always_comb sensors = { front_door, rear_door, window } ;

// procedural block for incrementing the state machine
always_ff @ ( posedge clk )
    if (reset)
        curr_state <= disarmed ;
    else
        curr_state <= next_state ;

// procedural block to determine the next state
always_comb begin
    unique case ( curr_state )
        disarmed: begin
            if ( keypad == 4'b0011 )
                next_state <= armed;

```

```
        else
            next_state <= curr_state ;
        end
    armed: begin
        if ( sensors != 3'b000 )
            next_state <= wait_delay;
        else if ( keypad == 4'b1100)
            next_state <= disarmed;
        else
            next_state <= curr_state ;
        end
    wait_delay: begin
        if (count_done == 1'b1)
            next_state <= alarm;
        else if ( keypad == 4'b1100 )
            next_state <= disarmed ;
        else
            next_state <= curr_state ;
        end
    alarm: begin
        if ( keypad == 4'b1100 )
            next_state <= disarmed;
        else
            next_state <= curr_state ;
        end
    endcase
end

// procedural block to generate the state machine output values
always_ff @ ( posedge clk ) begin
    if (reset) begin
        is_armed          <= 1'b0 ;
        is_wait_delay <= 1'b0 ;
        alarm_siren      <= 1'b0 ;
    end
    else
        begin
            is_armed          <= ( next_state == armed );
            is_wait_delay <= ( next_state == wait_delay );
            alarm_siren      <= ( next_state == alarm );
        end
    end
end
```

```
    always_comb start_count = (( curr_state == armed) && (sensors != 3'b000));  
  
    // Implement the delay counter.  
    // Loads delay_cntr with delay_val-1 when start_count is high, then counts down to 0 and stops.  
    // The condition delay_cntr = 0 triggers the next state transition in the main state machine  
  
    always_ff @ ( posedge clk) begin  
        if (reset)  
            delay_cntr <= 0;  
        else if (start_count)  
            delay_cntr <= delay_val - 1'b1;  
        else if (curr_state != wait_delay)  
            delay_cntr <= 0;  
        else if (delay_cntr != 0)  
            delay_cntr <= delay_cntr - 1'b1;  
        end  
  
        always_comb count_done = (delay_cntr == 0);  
    endmodule
```


Lab 5: Functions, Tasks, and Packages

2021.1

Abstract

This lab introduces functions, tasks, and packages in SystemVerilog. In this lab, you will create a new package and add global declarations such as enumerated variables, structures, unions, functions, or tasks to that package. You will then import this package to your main module and perform arithmetic and logic operations.

This lab should take approximately 45 minutes.

Objectives

After completing this lab, you will be able to:

- Describe what a package is and how it is used
- Write code in SystemVerilog to create a package
- Import all the global variables, structures, unions, functions, or tasks that are declared in a package to the module
- Simulate and synthesize a SystemVerilog design using the Vivado® Design Suite

Introduction

The SystemVerilog model functionality used in this lab works as an 8-bit miniature ALU. It does either four arithmetic or four logical operations on 8-bit operands.

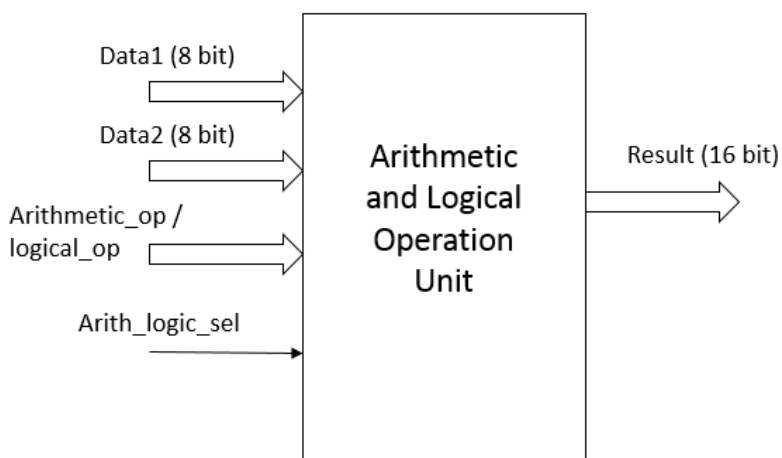


Figure 5-1: Block Diagram

In this lab, you will create an `arith_logic_pkg` package file and include all the declarations in it (such as enumerated data types, structures, unions, functions or tasks, etc.). The advantage of using a package is that it allows you to define commonly used variables in the package and then reuse them in any module at a later time.

This package contains the enumerated type definitions for `arithmetic_operation` and `logical_operation`. The `arithmetic_op` variable selects between four available arithmetic operations: addition, subtraction, multiplication, and division. Similarly, the `logical_op` variable decides between four logical operations that are available: `nand`, `nor`, `not`, and `xor`.

The package also has two structures: `packed_arith_logic_info` and `unpacked_arith_logic_result`. The arithmetic or logical operations are done on 8 bits `data1` and `data2` signals (specified in `arith_logic_info`) and are accessed with the help of the `arith_data` and `logic_data` object.

The final result is of 16 bits and is stored in the packed union called `result`. This union is defined inside an unpacked structure called `arith_logic_result`.

These arithmetic operations are done in different functions and tasks. The addition and subtraction operations are done in the functions named `addition` and `subtraction`, respectively. The multiplication and division operations are done in the tasks named `multiplication` and `division`, respectively. The two parameters need to be passed to these functions on which the operations are done. The function returns its result in the variable which has same name as the function, whereas the task has a specific output port where the result is stored.

You will find the following signals in this module:

- `arith_logic_sel` – Selects between arithmetic and logical operations.
- `operation` – Selects which arithmetic or logical operations is to be done.
- `ip_data1` and `ip_data2` – Two 8-bit data inputs.
- `data_out` – Output of the selected operation.

These signals are described in the package, and this package is then imported to the main module and the functionality is achieved.

Understanding the Lab Environment

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

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

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

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

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

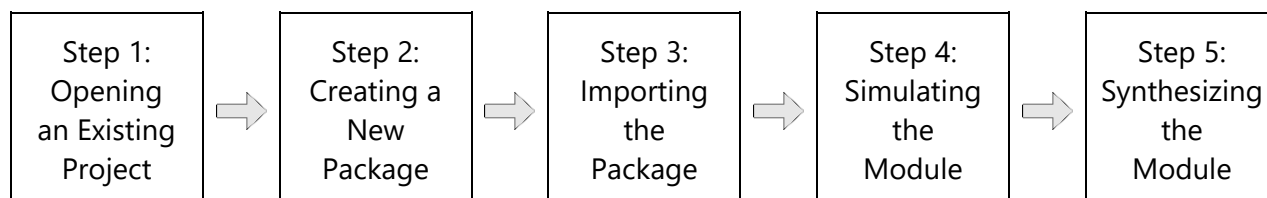
Environment Variable Name	Description
<code>\$TRAINING_PATH</code>	<p>Points to the space allocated for students to work through the labs. This directory includes prebuilt images and starting points for the labs and demos.</p> <p>The customer training VM sets <code>\$TRAINING_PATH</code> to the <code>/home/xilinx/training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

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

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

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

General Flow



Opening an Existing 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. Open the Vivado Design Suite project named `package.xpr` located in the directory below.

1-2-1. Browse to the `$TRAINING_PATH/sv_package/lab/KCU105` directory and open the `package.xpr` project.

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

Creating a New Package

Step 2

In this step, you will open the `arith_logic.sv` file in the project and create a new SystemVerilog package called `arith_logic_pkg`. You will then copy all the user defined enums, structures, and unions from `arith_logic module` into this package and also add two new functions and two new tasks to this package.

2-1. Open the `arith_logic` module.

2-1-1. Expand **Design Sources** in the Sources window and double-click **arith_logic**.

Observe the user-defined enums, structures, and unions that were mentioned in the Introduction to this lab.

2-2. Create a new HDL source file called `arith_logic_pkg`.

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

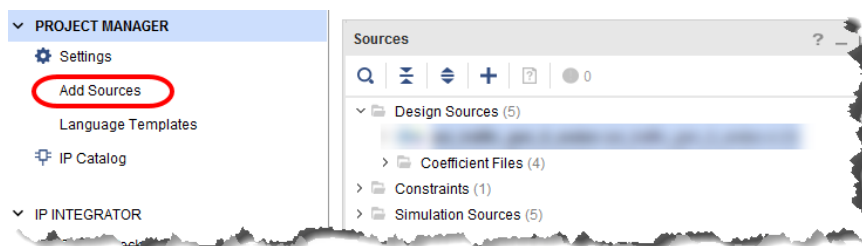
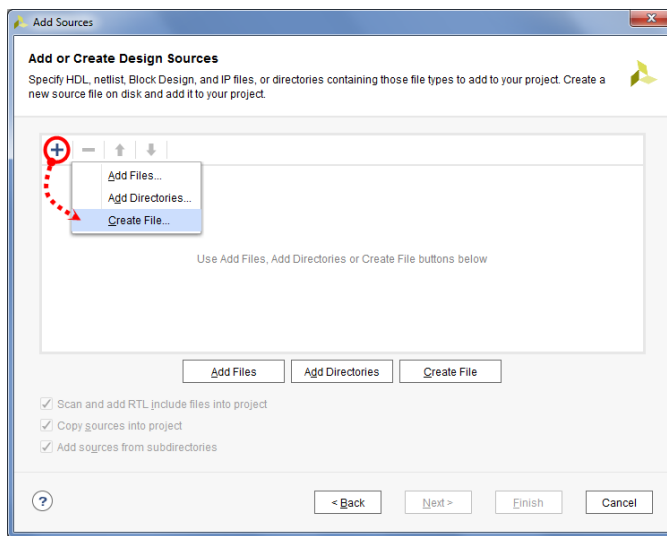


Figure 5-2: Selecting Add Sources

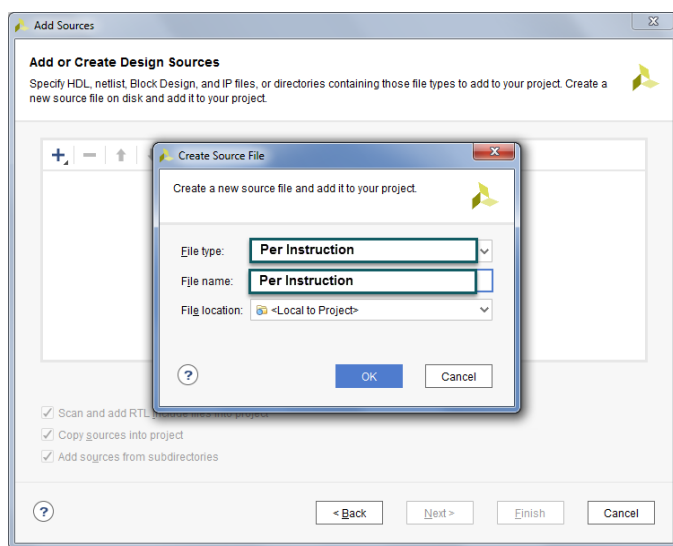
2-2-2. Select Add or create design sources (1).**Figure 5-3: Selecting Add or Create Design Sources****2-2-3. Click Next (2).**

The Add or Create Design Sources dialog box opens.

2-2-4. Click the Plus (+) icon and select Create File.**Figure 5-4: Selecting Create File**

The Create Source File dialog box opens.

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

2-2-6. Enter `arith_logic_pkg` as the file name.**Figure 5-5: 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.

2-2-9. Click **OK to create a module without ports.****2-2-10. Click **Yes** to confirm that the module definition has not been changed.****2-3. Open the newly created SystemVerilog file for editing.****2-3-1. Expand **Design Sources** from the sources window.****2-3-2. Double-click `arith_logic_pkg`.**

The SystemVerilog file opens for editing.

2-4. Copy the code from `arith_logic` to the newly created `arith_logic_pkg`.**2-4-1. Select the `arith_logic` tab to view the file.****2-4-2. Copy only the user-defined enums, structures, and unions.****2-4-3. Select the `arith_logic_pkg` tab.****2-4-4. Select all (<Ctrl + A>), and press <Delete> to clear the contents of `arith_logic_pkg`.****2-4-5. Add the keywords package and endpackage.****2-4-6. Right-click and select **Paste** in between the two keywords.**

2-5. Add the functions and tasks to the SystemVerilog file.**2-5-1.** Add the addition and subtraction functions to the package.

These functions accept the 8-bits operands data1 and data2 as parameters, complete the appropriate operation, and return a 16-bit result.

Note that the return variable for the functions has the same name as the function.

2-5-2. Add the multiplication and division tasks to the package in the same way.

As per the previously created functions, these tasks also accept the 8-bit operands data1 and data2 as parameters, complete the appropriate operation and return a 16-bit result. Here, mult_result and div_result are the two output variables which stores the result.

Hint: Although addition and subtraction functions do not need a 16-bit result, multiplication may generate a result larger than 16-bits result, Because of this, the result variable is taken as 16 bits.

2-5-3. Ensure that your new package file contains the following:

- The two user-defined enumerated type declaration arith_operation and logic_operation
- The user-defined packed structure arith_logic_info
- The user-defined unpacked structure arith_logic_result with a packed union result
- Two functions addition and subtraction
- Two tasks multiplication and division

2-5-4. Select **File > Text Editor > Save File** to save the file.

Note: The completed code is available for your reference in the Answers section.

Importing the Package into the Module

Step 3

In this step, you will add and edit the `import_package` file here. You will import the package into this module and access the structures, unions, functions, and tasks which are inside a package. You will then perform the operations on the members as per the given inputs.

3-1. Add an HDL source file to the design.

3-1-1. Select **Add Sources** under the Flow Navigator tab, under Project Manager.

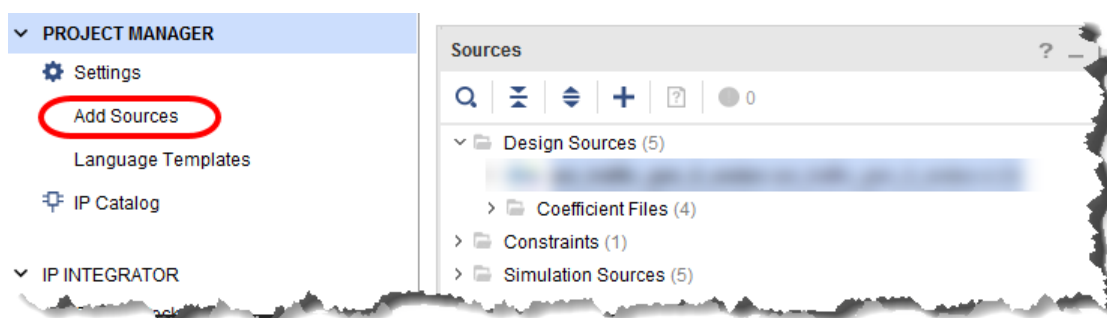


Figure 5-6: Selecting Add Sources

The Add Sources dialog box opens, allowing you to add HDL source files to the project.

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



Figure 5-7: Selecting Add or Create Design Sources

3-1-3. Click **Next** to begin selecting source files.

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.

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

3-1-5. Browse to \$sv_package/support/src.

3-1-6. Select **import_package.sv**.

3-1-7. Click **OK** in the Add Source Files dialog box to select the file(s).

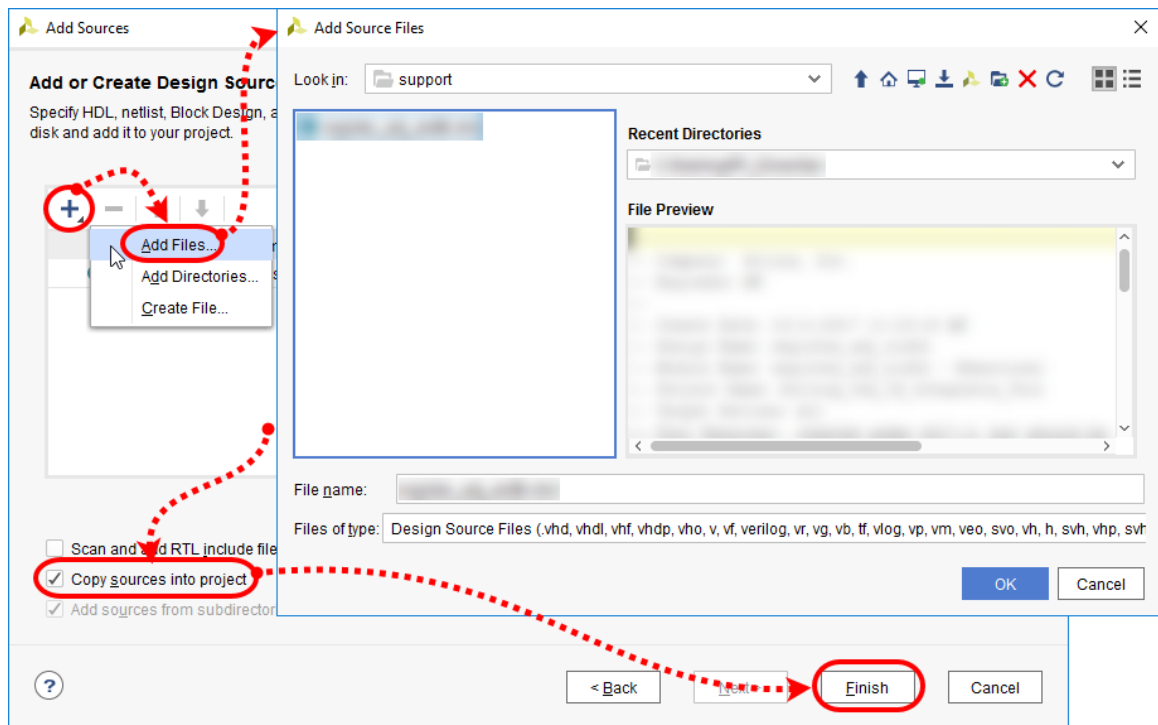


Figure 5-8: Selecting Add Files

3-1-8. Ensure that the **Copy sources into project** option is selected.

3-1-9. Click **Finish** in the Add or Create Design Sources dialog box to add the HDL sources to the project.

3-2. Open and edit an existing SystemVerilog module.

3-2-1. Expand **Design Sources > import_package** in the Sources window.

3-2-2. Double-click to open the file.

3-2-3. Right-click **import_package** and choose **Set as Top** to make it the top module.

3-2-4. Import the created package into this module with the help of the import statement given below.

```
import arith_logic_pkg :: *;
```

This will import the entire package into this module.

Note that the star (*) here indicates that the entire contents from the package will be imported. You can add this statement anywhere in the module except before the objects declaration.

- 3-2-5.** Observe the `arith_data` and `logic_data` objects that are created for the `arith_logic_info` structure and the `final_result` object for the `arith_logic_result` structure in the module.
- 3-2-6.** Observe how the objects created are used to access the members of the structures. The module has following inputs:

`arith_logic_sel` - Selects between arithmetic and logical operation

`operation` - 2-bit input, Selects between the either the arithmetic or logical operations

`ip_data1`, `ip_data2` - 8-bit data inputs

These inputs are assigned to the appropriate members, and operations are performed on the members of the structure and the result is stored in either the `arith_result` or `logic_result`, which are part of the union.

You can add the function calls for the addition and subtraction functions. They accept two parameters of 8 bits. Similarly, you can also add the task calls for the multiplication and division tasks. They accept two parameters of 8 bits and return a 16-bit result.

Note: The completed code is available for your reference in the Answers section.

- 3-2-7.** Select **File > Text Editor > Save File** to save the file.
- 3-2-8.** In the Sources window, Hierarchy tab, right-click **`arith_logic.sv`** under Design Sources and select **Disable File**.

3-3. Adding a new SystemVerilog testbench file called `tb_import_package`. HDL simulation files can be added to the design at any time.

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

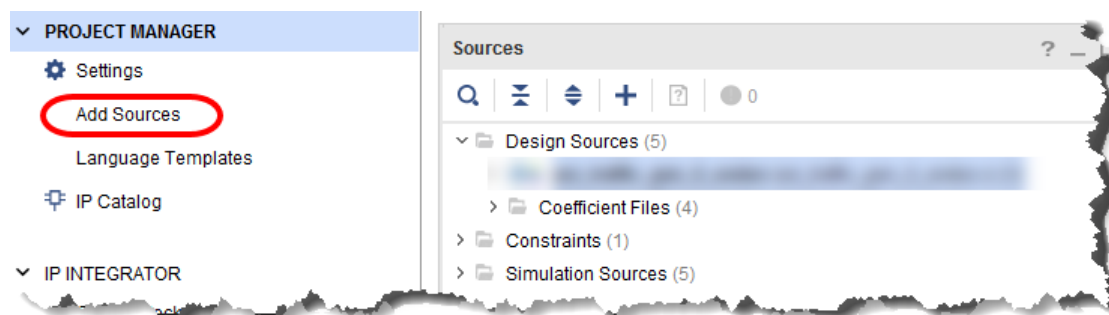
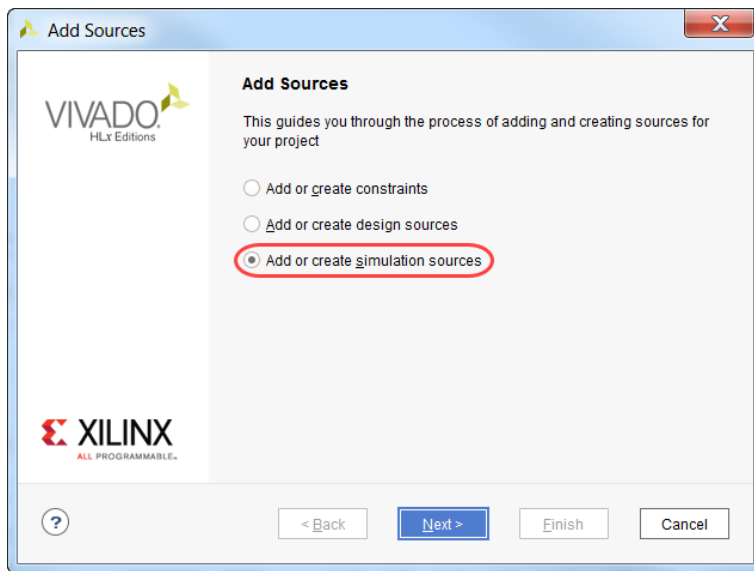


Figure 5-9: Selecting Add Sources

3-3-2. Select Add or create simulation sources.**Figure 5-10: Add Sources Dialog Box****3-3-3. Click Next.****3-3-4. Click the Plus (+) icon to open the pop-up menu.****3-3-5. Select Add Files** to open the Add Source Files dialog box which allows you to browse to the desired directory.**3-3-6. Browse to the \$TRAINING_PATH/sv_package/support/src** directory if it is not open already.**3-3-7. Select tb_import_package.sv.****3-3-8. Click OK** in the Add Source Files dialog box.**3-3-9. Click Finish** to add the file(s) to the project.**3-3-10. Open the file in text editor** and review the simple testbench which gives input stimulus.

Simulating the tb_import_package Module

Step 4

In this step, you will simulate the tb_import_package module and observe the outputs.

The following instructions are for Vivado simulator users only. Mentor Graphics Questa users can proceed to the instruction that begins with "The following instructions are for Questa users only".

4-1. Run the simulation.

- 4-1-1. Select **Simulation > Run Simulation > Run Behavioral Simulation** from the Flow Navigator.

The simulation runs for the default **1us**.

- 4-1-2. Use the zoom options in the toolbar to scale the waveform window.

4-2. Change the radix and verify the simulation result.

- 4-2-1. Verify that the arith_logic_sel signal in the waveform window decides between the arithmetic and logical operation.
- 4-2-2. Right-click the ip_data1, ip_data2, and data_out signals and select **Radix > Unsigned Decimal**.

This will make it easier to analyze the waveform while checking arithmetic operations.

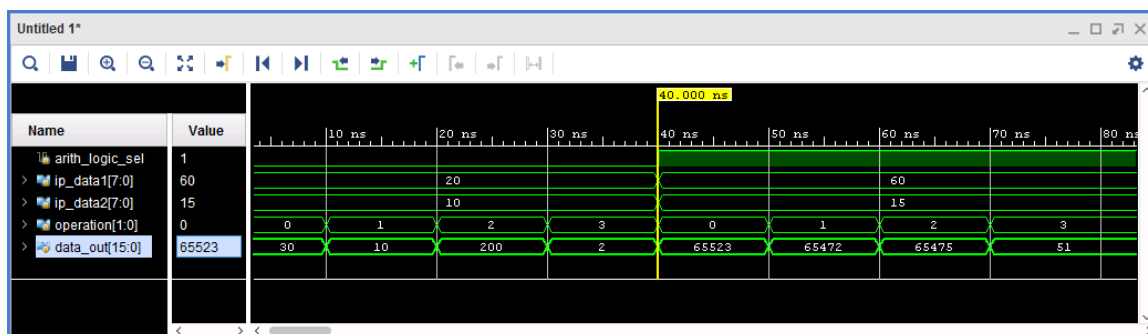


Figure 5-11: Simulation Results

The operation signal decides between the arithmetic operations, with operation = 0 being addition, operation = 1 being subtraction, and so on. Verify the data_out value as per the operation selected.

- 4-2-3.** Right-click the ip_data1, ip_data2, and data_out signals again and select **Radix > Binary**. This will make it easier to analyze the waveform while checking logical operations.

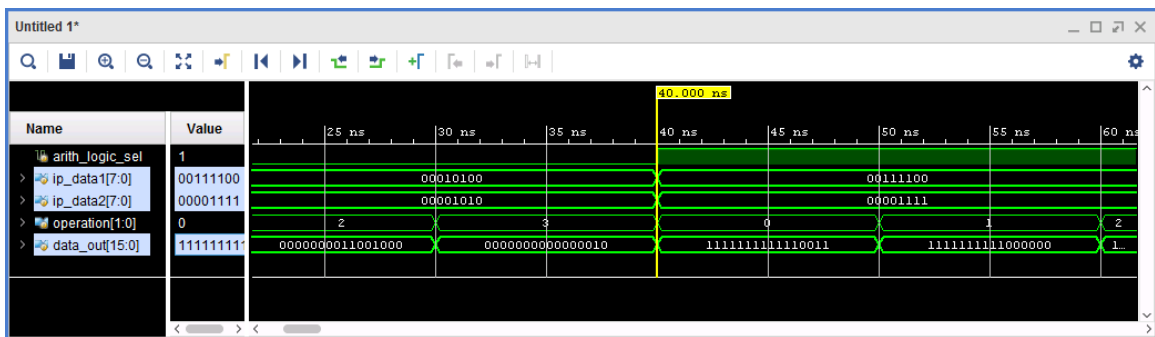


Figure 5-12: Simulation Results

The operation signal now decides between the logical operation, with operation = 0 being nand_op, operation = 1 being nor_op, and so on. Verify the data_out value as per the operation selected.

4-3. Exit the simulator.

- 4-3-1.** Select **File > Close Simulation**.
4-3-2. Click **OK** to confirm.
4-3-3. Click **Discard** to exit without saving the waveform.

The following instructions are for Questa users only. Vivado simulator users can proceed to the next step of this lab.

4-4. Set the simulation target to Questa Advanced Simulator.

- 4-4-1.** Select **Settings** under Flow Navigator and select **Simulation** under Project Settings.
4-4-2. Set the Target Simulator field to **Questa Advanced Simulator**.
4-4-3. Click **Yes** in the Target Simulation dialog box.

This changes the target simulator to **Questa Advanced Simulator**.

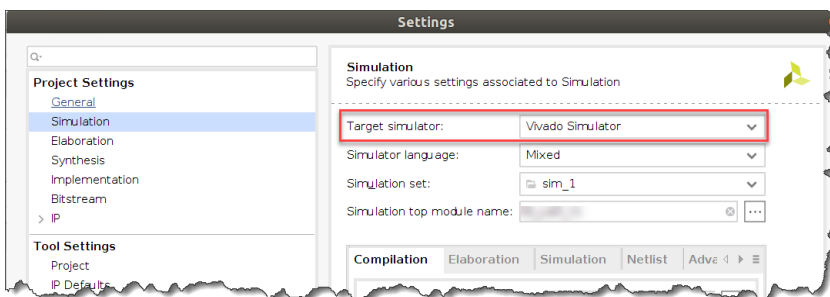


Figure 5-13: Simulation Settings

- 4-4-4.** Click **OK** in the Project Setting dialog box.

4-5. Run the simulation.

4-5-1. Select **Simulation > Run Simulation > Run Behavioral Simulation** from the Flow Navigator.

4-5-2. Use the zoom options in the toolbar to scale the waveform window.

4-6. Change the radix and verify the simulation result.

4-6-1. Verify that the arith_logic_sel signal in the waveform window decides between the arithmetic and logical operation.

4-6-2. Right-click the ip_data1, ip_data2, and data_out signals and select **Radix > Decimal**.

This will make it easier to analyze the waveform while checking arithmetic operations.

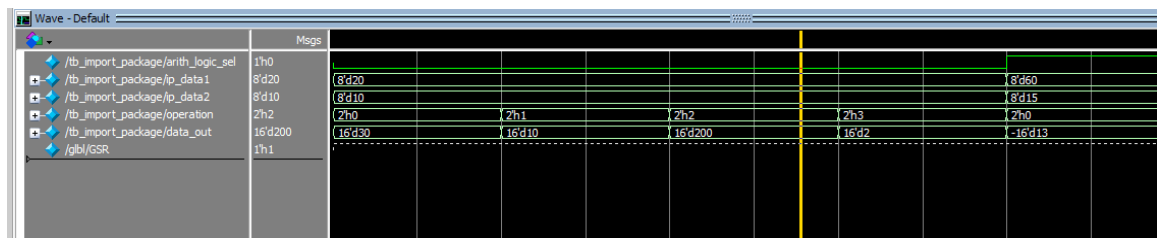


Figure 5-14: Simulation Results

The operation signal decides between the arithmetic operations, with operation = 0 being addition, operation = 1 being subtraction, and so on. Verify the data_out value as per the operation selected.

4-6-3. Right-click the ip_data1, ip_data2, and data_out signals again and select **Radix > Binary**.

This will make it easier to analyze the waveform while checking logical operations.

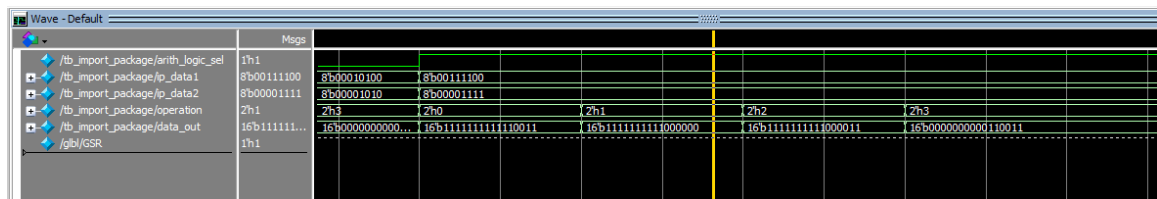


Figure 5-15: Simulation Results

The operation signal now decides between the logical operations, with operation = 0 being nand_op, operation = 1 being nor_op, and so on. Verify the data_out value as per the operation selected.

4-7. Exit the simulator.

4-7-1. Select **Simulate > End Simulation** to close the simulator.

4-7-2. Click **Yes** to confirm.

4-8. Close Questa Sim.

4-8-1. Select **File > Quit**.

4-8-2. Click **Yes** to close Questa Sim.

Synthesizing the import_package Module**Step 5**

In this step, you will synthesize the import_package SystemVerilog design to show how packages are imported to the main module and how the module can be synthesized by the Vivado Synthesis tool.

5-1. Explore the synthesis settings available.

5-1-1. Click **Settings** in the Flow Navigator and select **Synthesis** under Project Settings.

Note that the strategy is set to **Vivado Synthesis Defaults**.

5-1-2. Review the other synthesis options.

5-1-3. Click **OK** after reviewing the other options.

5-2. Run design synthesis based on the settings selected.

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

5-2-2. View any of the reports generated by selecting **View Reports** in the Synthesis Completed dialog box after synthesis completes successfully.

5-2-3. Click **Cancel** to close the Synthesis Completed dialog box.

5-3. Close the Vivado Design Suite.

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

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

5-4. [Optional] [Only for local VMs—not for CloudShare] Clean up the file system.

Using the GUI:

5-4-1. Using the graphical browser (Windows: press the <**Windows**> key + <**E**>; Linux: press <**Ctrl** + **N**>), navigate to \$TRAINING_PATH/sv_package.

5-4-2. Select **sv_package**.

5-4-3. Press **<Delete>**.

-- OR --

Using the command line:

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

5-4-5. Enter the following command to delete the contents of the workspace:

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

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

Summary

In this lab, you learned how to create a new package in SystemVerilog and how to import that package into the main module. You learned about the functions and tasks and their benefits. You used them to do arithmetic operations on the members of the structure.

Finally, you learned how to synthesize a package in SystemVerilog design with Vivado Design Suite.

Answers

arith_logic_pkg.sv

```
package arith_logic_pkg;

    // User defined enumerated data type declaration
    typedef enum logic[1:0] {add, sub, mul, div} arith_operation;
    typedef enum logic[1:0] {nand_op, nor_op, not_op, xor_op} logic_operation;

    // User defined Packed Structure declaration
    typedef struct packed {
        arith_operation arithmetic_op;
        logic_operation logical_op;
        logic [7:0] data1;
        byte data2;
    } arith_logic_info;

    // User defined Unpacked Structure declaration
    typedef struct {
        // User defined Packed Union declaration
        union packed {
            logic [15:0] arith_result;
            logic [15:0] logic_result;
        } result;
    } arith_logic_result;

    function [15:0] addition;
        input [7:0] data1, data2;
        begin
            addition = data1 + data2;
        end
    endfunction

    function [15:0] subtraction;
        input [7:0] data1, data2;
        begin
            subtraction = data1 - data2;
```

```
        end
    endfunction

    task multiplication;
        input [7:0] data1, data2;
        output [15:0] mult_result;
        begin
            mult_result = data1 * data2;
        end
    endtask

    task division;
        input [7:0] data1, data2;
        output [15:0] div_result;
        begin
            div_result = data1 / data2;
        end
    endtask
endpackage
```

import_package.sv

```
`timescale 1ns / 1ps
import arith_logic_pkg :: *;

module import_package (    input logic arith_logic_sel,                // Selects
    between arith or logic operations
                                input logic [1:0] operation,          // Selects which arith or logic
    operations is to be done
                                input logic [7:0] ip_data1, ip_data2,  // Two data inputs
                                output logic [15:0] data_out);          // Output of the selected
    operation

    always @ (*)
    begin
```

```

        arith_logic_info arith_data, logic_data;           // Creating two objects arith_data
and logic_data
        arith_logic_result final_result;                 // Creating an object of structure
arith_logic_result
        if (arith_logic_sel==0)
        // Selecting Arithmetic operation
        begin
                arith_data.arithmetic_op = arith_operation'(operation);
                arith_data.logical_op = logic_operation'(0);
                arith_data.data1 = ip_data1;
                arith_data.data2 = ip_data2;

        case (arith_data.arithmetic_op)                   // Accessing enum data type
through arith_data input
        // As per the value of enum, do the arithmetic operations
        // Accesing the Packed Structure for two datas and storing the result in an Packed
Union
        // Using two functions for addition and subtraction
        add : final_result.result.arith_result = addition (arith_data.data1,
arith_data.data2);           // function call
        sub : final_result.result.arith_result = subtraction (arith_data.data1,
arith_data.data2);           // function call
        // Using two tasks for multiplication and division
        mul : multiplication (arith_data.data1, arith_data.data2,
final_result.result.arith_result); // task call
        div : division (arith_data.data1, arith_data.data2,
final_result.result.arith_result); // task call
        endcase
        data_out = final_result.result.arith_result;

    end

    else

```

```
// Selecting Logical operation
begin
    logic_data.arithmetic_op = arith_operation'(0);
    logic_data.logical_op = logic_operation'(operation);
    logic_data.data1 = ip_data1;
    logic_data.data2 = ip_data2;

    case (logic_data.logical_op)          // Accessing enum data type through
logic_data input
    // As per the value of enum, do the logic operations
    // Accesing the Packed Structure for two datas and storing the result in an Packed
Union
        nand_op : final_result.result.logic_result = ~((logic_data.data1) &
(logic_data.data2));
        nor_op  : final_result.result.logic_result = ~((logic_data.data1) |
(logic_data.data2));
        not_op   : final_result.result.logic_result = ~ (logic_data.data1);
        xor_op   : final_result.result.logic_result = ((logic_data.data1) ^
(logic_data.data2));
    endcase
    data_out = final_result.result.logic_result;
end
end
endmodule
```


Lab 6: Interfaces and Design Download

2021.1

Abstract

This lab guides you through the process of using interface modports to connect blocks in a design. You will download the design onto a Kintex® UltraScale™ KCU105 evaluation board and verify the functionality of the design.

This lab should take approximately 60 minutes.

Objectives

After completing this lab, you will be able to:

- Create and use SystemVerilog interfaces
- Use interface modports across hierarchy levels
- Run implementation and generate a bitstream in the Vivado® Design Suite
- Download a design onto a Kintex UltraScale KCU105 evaluation board

Introduction

Interfaces in SystemVerilog:

- An interface is an abstract port type that allows many signals to be grouped together and represented as a single port.
- Interfaces are synthesizable.
- Modports provide the ability to use the interface signals differently at each module interface port.
- A modport defines the port direction that the module sees for the signals in the interface.

The *uart_led* design used in this lab receives data entered by the user in serial format and outputs eight bits that are sent to the LEDs.

The design primarily consists of two blocks:

- A UART receiver that receives RS-232 serial data at 115200 baud (no parity, 8 data bits, no handshaking):
 - Handled in *uart_rx* - simple state machine and an over sampler.
 - The over sampler takes 16 samples per bit and uses this information to sample in the middle of each actual "bit".
 - An LED controller that creates the binary equivalent of a successfully received character (intended for showing up on LEDs).

- The top-level design (*top.sv*) that enables the proper functioning of the *uart_led* design has the following block diagram:

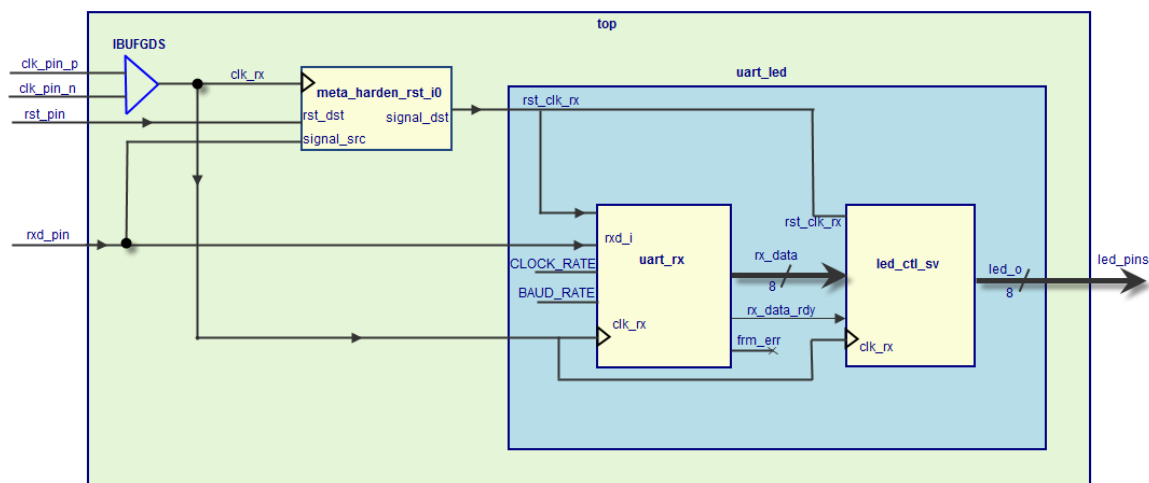


Figure 6-1: Top-Level Block Diagram of the *uart_led* Design

The differential clock is buffered by an IBUFGDS to bring the signal into the FPGA. A pushbutton reset is fed to the *meta_harden* module where the risk of becoming metastable is greatly reduced. This “cleaned” reset signal is provided to the *uart_led* module.

The *uart_rx* captures serial data (*rxd_pin*) and presents it at the output of the module on the *rx_data* 8-bit bus. The signal *rx_data_rdy* pulses high for one clock when a new *rx_data* is received. Note that the framing error signal from *uart_rx* is not used in this design.

This assertion of *rx_data_rdy* activates the *led_ctl* module, which captures the received character and holds it for display on the LEDs. Finally, the output from the *led_ctl* module is driven to the package pins via eight OBUFs.

CLOCK_RATE is a generic/parameter that is specified in Hz. This value is used during elaboration/compilation to compute various settings which rely on the clock frequency.

BAUD_RATE is a generic/parameter that is specified in baud. This value is used during elaboration/compilation to divide the clock frequency to generate the selected baud rate.

The *uart_rx* module hardens the incoming serial signal against metastability and passes the signal to the *uart_rx_ctl* module.

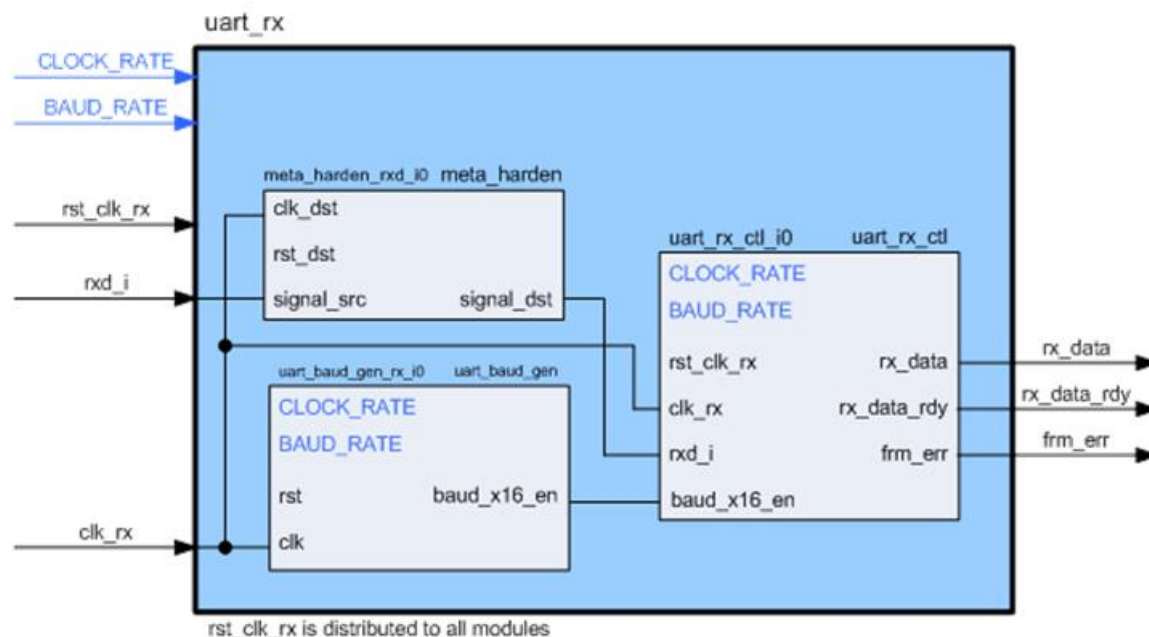


Figure 6-2: uart_rx Module

The *uart_rx_ctl* module is a state machine. The IDLE state waits for a falling edge on the incoming signal. After this edge is detected, it begins counting eight occurrences of *baud_x16_en* and looks at the hardened serial input line. This time delay should sample near the midpoint of the serial bit. If it is low, it assumes that this is a start bit and proceeds to collect the data bits; otherwise, it is assumed that the low-going value on the serial line was a glitch and returns to the IDLE state.

Data is collected in a similar fashion - 16 samples are counted and the serial line is sampled, and that sample is shifted into a holding register. When all the data has been collected, the serial line is once again sampled (16 *baud_x16_en* occurrences later, or in the middle of the stop bit).

Once the stop bit has been sampled, then the captured data is presented in a parallel fashion on *rx_data* and *rx_data_rdy* is pulsed for one clock cycle to indicate that new data has arrived. In the event that the final sample (the stop bit) was not high, *frm_err* signals is also pulsed to indicate a framing error.

Understanding the Lab Environment

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

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

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

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

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

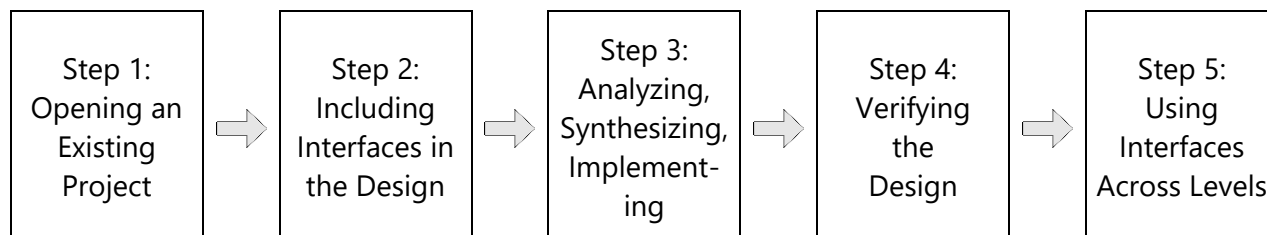
Environment Variable Name	Description
<code>\$TRAINING_PATH</code>	<p>Points to the space allocated for students to work through the labs. This directory includes prebuilt images and starting points for the labs and demos.</p> <p>The customer training VM sets <code>\$TRAINING_PATH</code> to the <code>/home/xilinx/training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

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

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

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

General Flow



Opening an Existing 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. Open the Vivado Design Suite project named `interfaces.xpr` located in the directory below.

1-2-1. Browse to the `$TRAINING_PATH/sv_interfaces/lab/KCU105` directory and open the `interfaces.xpr` project.

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

1-3. Analyze the source files.

1-3-1. Open the SystemVerilog modules corresponding to `uart_rx_ctl`, `uart_rx`, `led_ctl`, `uart_led`, and `top` from the Sources tab.

1-3-2. Analyze the code to ensure that it matches the design description given in the Introduction section.

Including Interfaces in the Design

Step 2

You will create interface modports that will be used by the `uart_rx` and `led_ctl_sv` modules.

2-1. Create a new HDL source file called `ifc_signals`.

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

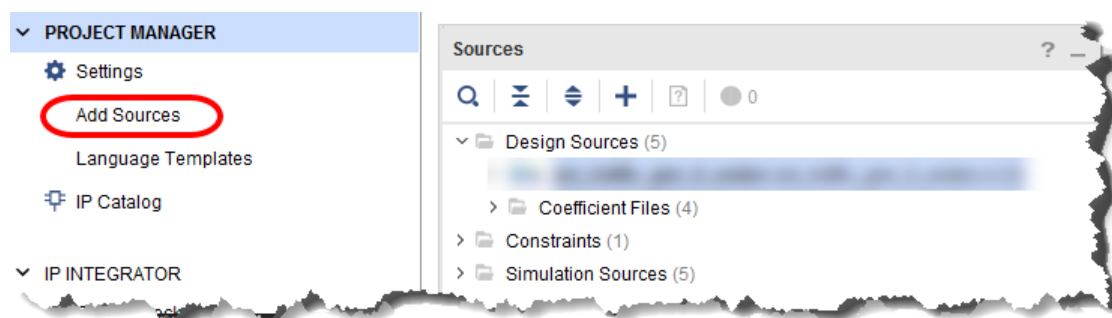
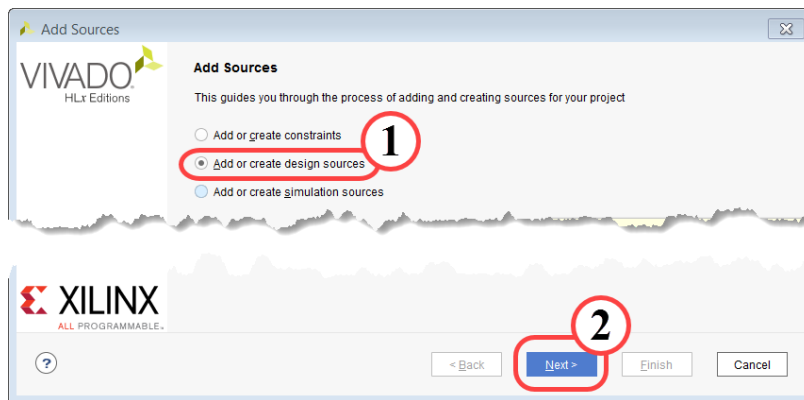
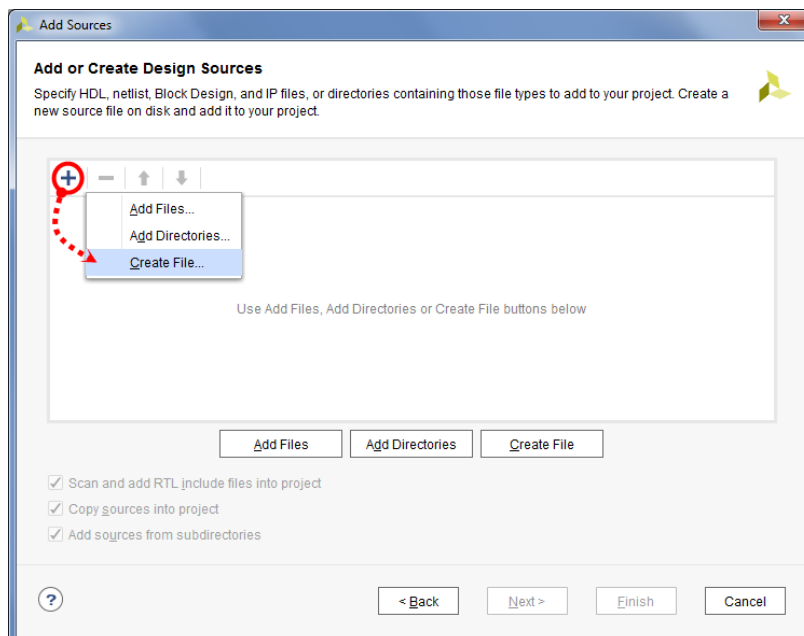


Figure 6-3: Selecting Add Sources

2-1-2. Select Add or create design sources (1).**Figure 6-4: Selecting Add or Create Design Sources****2-1-3. Click Next (2).**

The Add or Create Design Sources dialog box opens.

2-1-4. Click the Plus (+) icon and select Create File.**Figure 6-5: Selecting Create File**

The Create Source File dialog box opens.

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

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

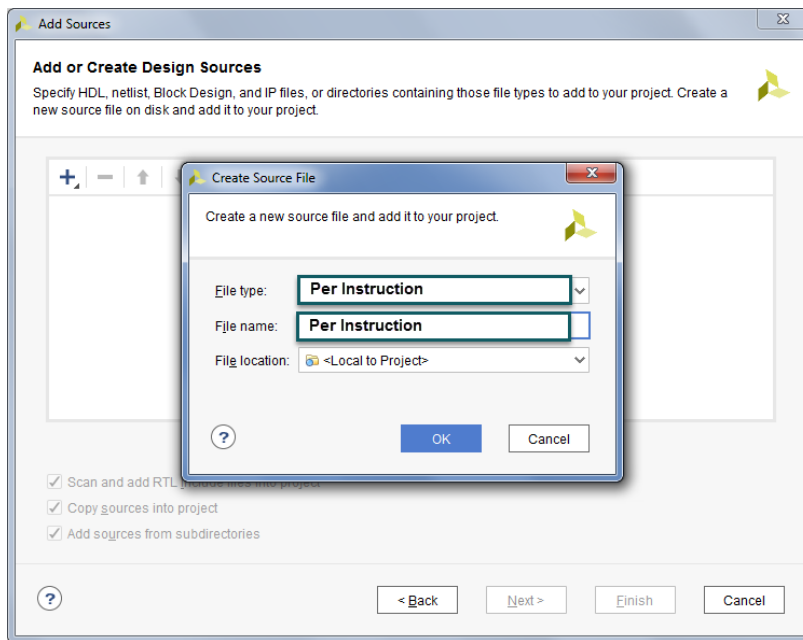


Figure 6-6: Entering File Name and Type

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

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

The Define Module dialog box opens.

2-1-9. Click **OK** to create the module without ports, since this module does not require them.

2-1-10. Click **Yes** to confirm that the module definition has not been changed.

2-1-11. Double-click **ifc_signals.sv** in the Libraries windows under Design Sources > SystemVerilog > xil_defaultlib > Unreferenced.

The SystemVerilog file opens for editing.

2-1-12. Remove the module definition.

This file will be used to define an interface.

2-2. Define the interface and modports.

2-2-1. Determine which signals in the *uart_led* design can be bundled in to an interface.

Hint: Observe the ports that are common to *uart_rx* and *led_ctl*.

2-2-2. Determine which of these signals should be declared as interface ports.

Recall that interface ports are used to bring in external signals in to the interface; i.e., signals that are ports to other modules.

2-2-3. Define an interface called *ifc_signals* in the *ifc_signals.sv* file.

2-2-4. Declare the interface ports and the internal signals of the interface.

2-2-5. Determine how many modports are required.

Hint: Observe the direction of the interface signals in *uart_rx* and *led_ctl*.

2-2-6. Define the modports in the *ifc_signals.sv* file.

Note: The direction of the interface ports will remain the same for all modports. Hence, it is not required to include the interface ports in the modports.

2-2-7. Select **File > Text Editor > Save File** to save the **ifc_signals.sv** file.

2-3. Include the interface modports in the design.

2-3-1. Open the **uart_rx** and **led_ctl** modules that will use the interface modports.

2-3-2. Replace the ports common to both *uart_rx* and *led_ctl* in the *uart_rx* module with the appropriate interface modport.

Hint: The syntax for using an interface modport as a module port is
<interface_name>.<modport_name> <modport_instance_name>.

In the body of the *uart_rx* module, rename the ports that have been replaced by the modport; i.e., replace all occurrences of <original_port_name> with
<modport_instance_name>.<original_port_name>.

Example: Replace all occurrences of *clk_rx* with <modport_instance_name>.*clk_rx*.

2-3-3. Select **File > Text Editor > Save File** to save the **uart_rx** file.

2-3-4. Update the *led_ctl* module in the same way.

2-3-5. Open the **uart_led** module that instantiates both the *uart_rx* and *led_ctl* modules.

2-3-6. Instantiate the interface (*ifc_signals*) and do a named connection of the interface ports with the *uart_led* ports.

Syntax: <interface_name> <top-level_interface_instance_name> (.<interface_port>
(<module_port>));

Example: *ifc_signals* *uart_led_ifc* (.*interfacePort1* (*clk_rx*), *interfacePort2* (*rst_clk_rx*));

2-3-7. Update the module instantiation of `uart_rx` appropriately.

Hint: The syntax for connecting an interface is `<module_name> <instance_name> (.<modport_instance_name> (<top-level_interface_instance_name>)`.

2-3-8. Update the instantiation of `led_ctl` in the same way.

2-3-9. Select **File > Text Editor > Save file** to save the `uart_led.sv` file.

2-3-10. Use the following example to verify your code for including interfaces.

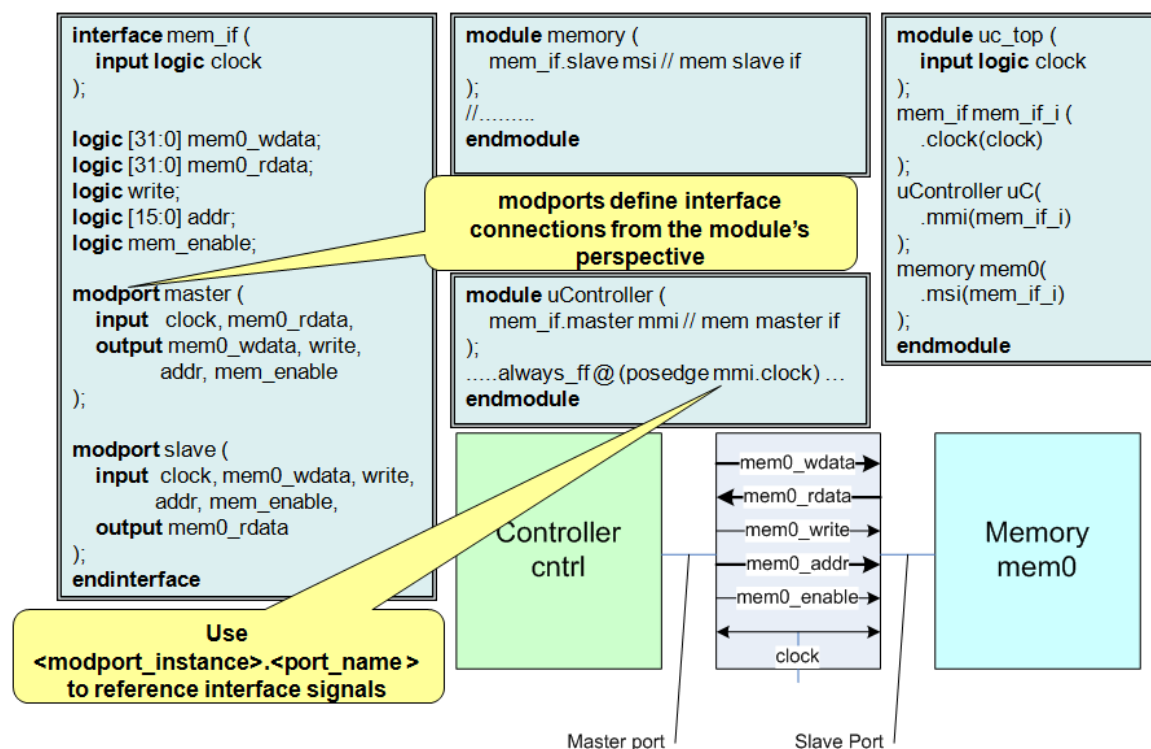


Figure 6-7: Interface modports

Analyzing the Schematic, Synthesizing and Implementing the Design

Step 3

In this step, you will examine the RTL schematic. Then you will synthesize and implement the design.

3-1. View and analyze the elaborated schematic.

3-1-1. Select **RTL Analysis > Open Elaborated Design** under the Flow Navigator and click **OK**.

This will elaborate the RTL sources in the design and display the RTL schematic for the top-level module in the design.

If the RTL schematic is not displayed, select **RTL Analysis > Open Elaborated Design > Schematic**.

3-1-2. Double-click the **uart_led** module to descend into it.

The RTL schematic of the *uart_led* module should appear as shown in the figure below.

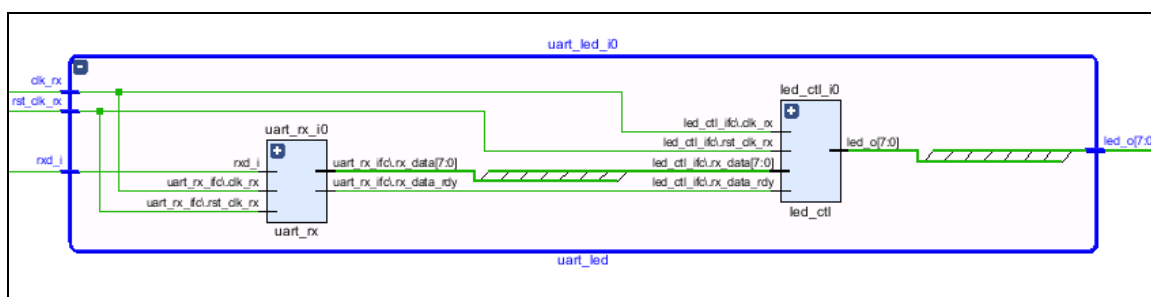


Figure 6-8: RTL Schematic of *uart_led*

Note the connections of the interface signals. In the above figure, the modport instance name used in *led_ctl* is *led_ctl_ifc*. The modport instance name used in *uart_rx* is *uart_rx_ifc*. The modport instance names may be different in your case and will reflect what was entered in your code.

Ensure that no ports are grounded. Grounded ports imply that the interface was not connected properly.

3-1-3. Double-click the **uart_rx** module to descend into it.

You should see an RTL schematic similar to the figure below.

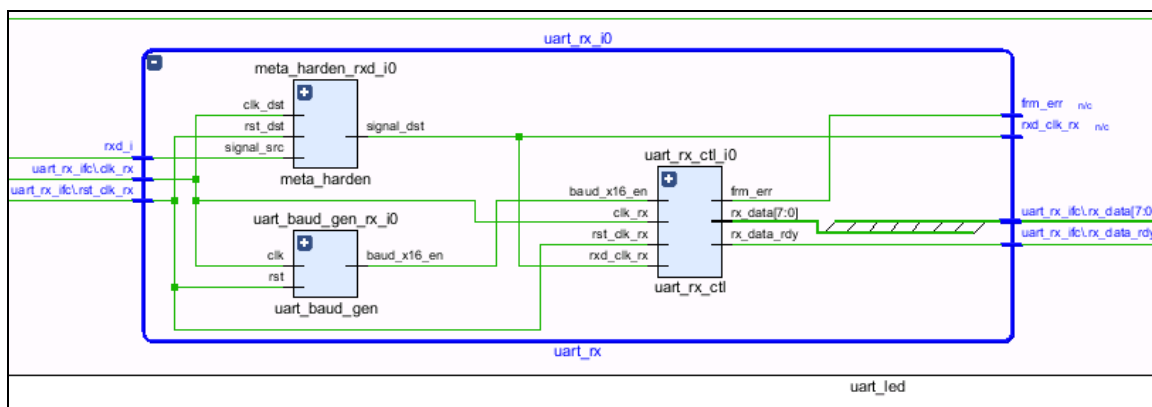


Figure 6-9: RTL Schematic of `uart_rx`

Again, ensure that the connections are correct and that the ports are not grounded.

3-1-4. Close the elaborated design once you are confident that the schematic is correct.

3-2. Run synthesis and implementation.

3-2-1. Select **Synthesis > Run Synthesis** under the Flow Navigator.

Resolve any errors that are reported.

3-2-2. Select **Run Implementation** in the Synthesis Completed dialog box.

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

This process will take few minutes.

3-2-4. Click **Cancel** in the Implementation Completed dialog box.

Verifying the Design on the Board

Step 4

In this step, you will perform bit file generation and then download the design to the Kintex UltraScale FPGA evaluation board. You will verify that the design works on the board as expected.

4-1. Generate the bitstream.

4-1-1. Click **Generate Bitstream** from the Flow Navigator, under Program and Debug.

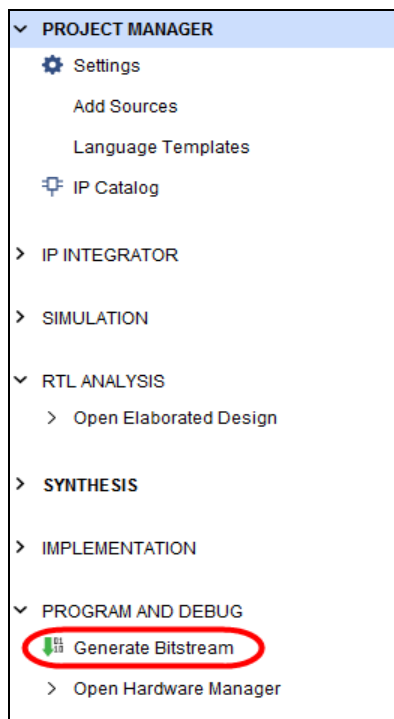


Figure 6-10: Generate Bitstream

4-1-2. Click **OK** to launch the runs.

Note that the Generate Bitstream process will try to resynthesize and implement the design if any process is out of date.

4-1-3. Click **Yes** if prompted to rerun any of the processes that are needed for bitstream generation.

4-1-4. Click **Cancel** in the Bitstream Generation Completed dialog box.

4-2. Power on the board.

- 4-2-1. Connect a USB cable from a USB port on your computer to the USB UART connector on the evaluation board.
- 4-2-2. Connect the USB cable to the Digilent USB JTAG interface.
- 4-2-3. Ensure that the power cord is plugged in and turn on the evaluation board.
- 4-2-4. Make sure that the board settings are proper.

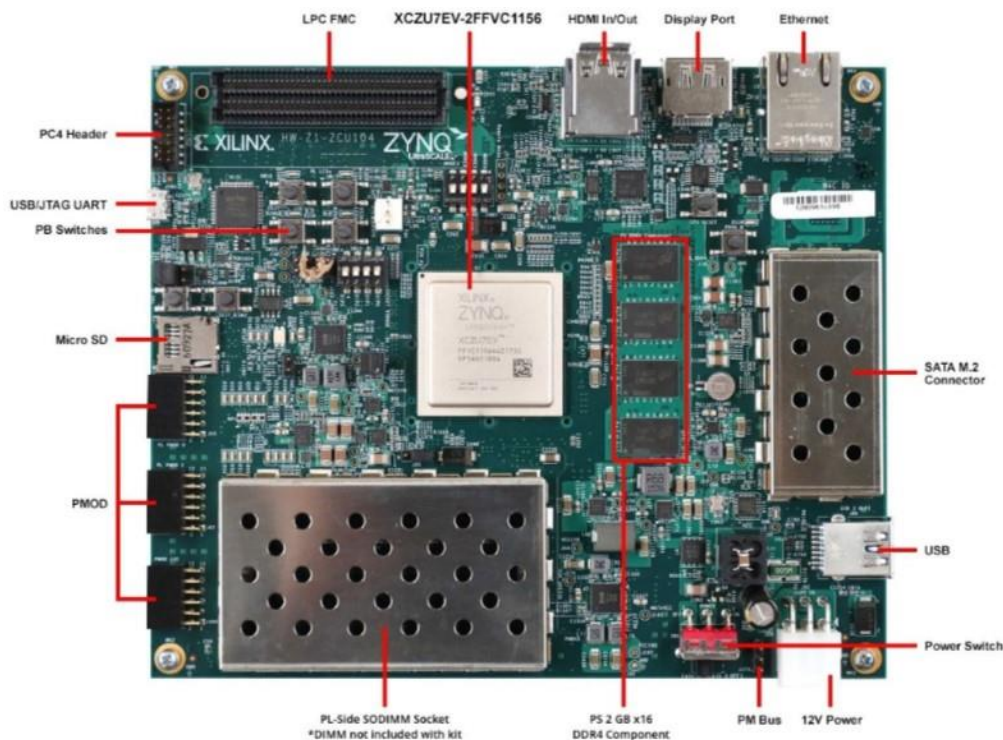


Figure 6-11: ZCU104 Evaluation Board

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.

A hardware manager is the portion of the Vivado Design Suite that enables the monitoring of cores that were added to a design.

4-3. Open the hardware manager.

4-3-1. 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.

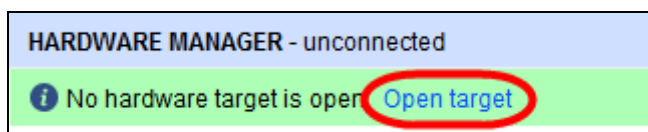


Figure 6-12: Opening a Hardware Target

4-4. Connect the target through the New Target Wizard to guide you through the process.

4-4-1. Click **Open target** > **Open New Target**.

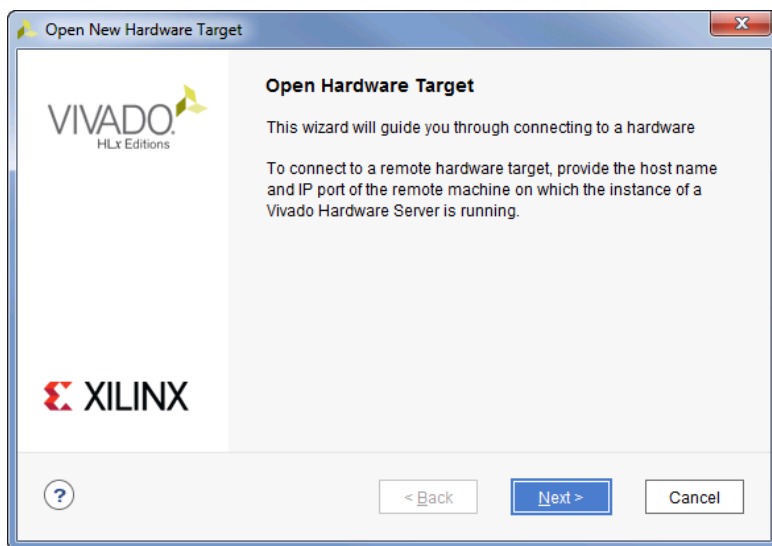


Figure 6-13: Open Hardware Target Dialog Box

4-4-2. Click **Next** to set the hardware server settings.

4-4-3. Enter a name for the server.

Typically, this is left at its default value.

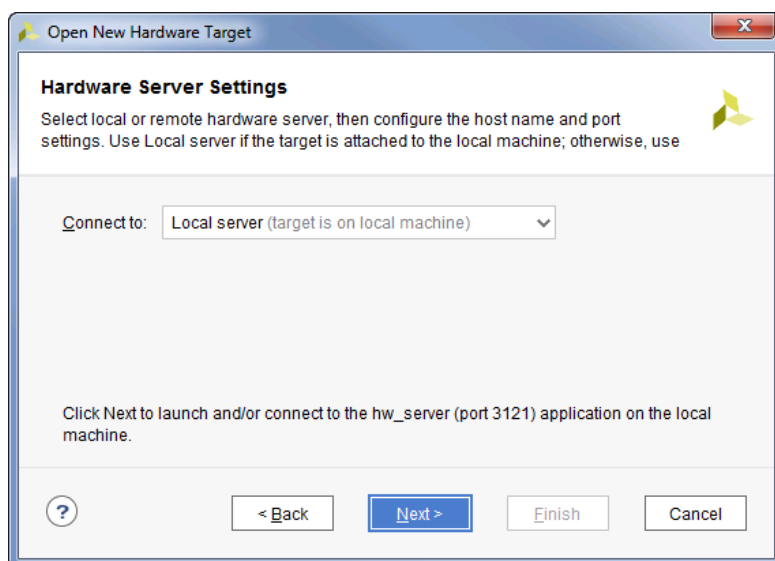


Figure 6-14: Setting the Server Name for the New Hardware Target

4-4-4. Click **Next** to select the hardware target.

4-4-5. 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.

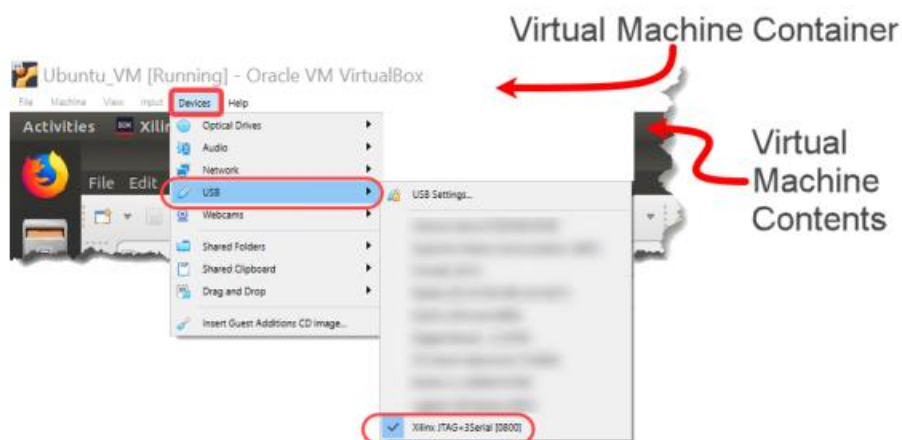


Figure 6-15: Selecting the Hardware Target

4-4-6. Leave the frequency at its default value.

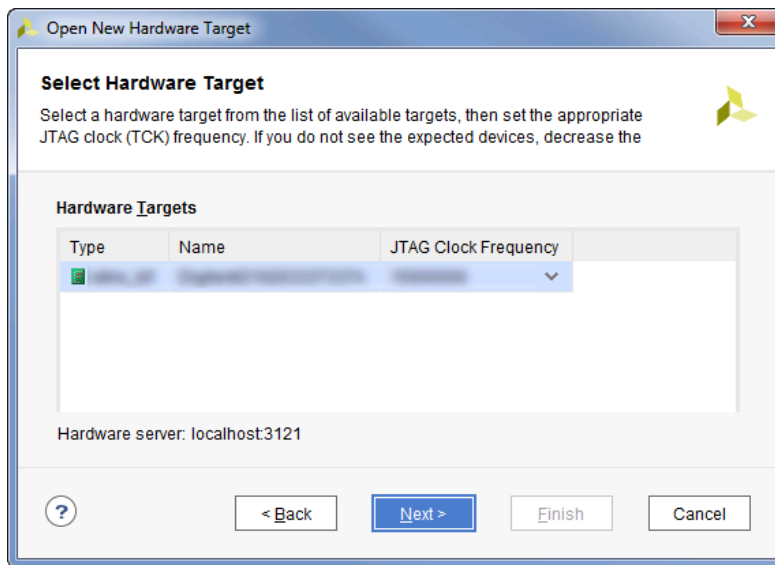


Figure 6-16: Selecting the Hardware Target

4-4-7. Click **Next** to view the hardware target summary.

A summary of the connection is displayed.

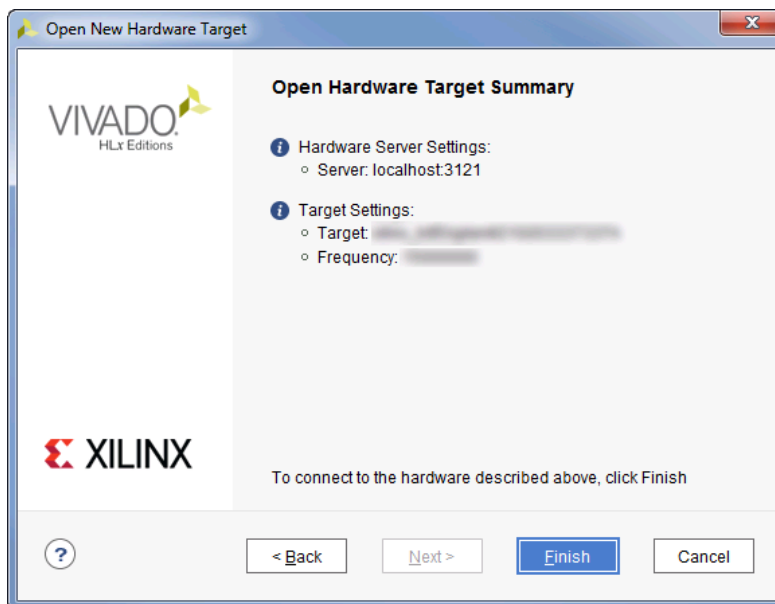
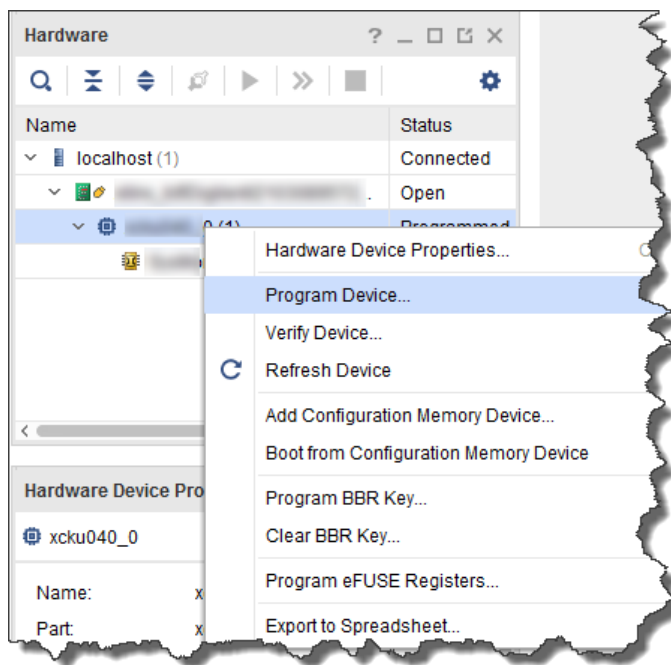


Figure 6-17: Summary of the Open Hardware Target Settings

4-4-8. Click **Finish** to connect to the new hardware target.

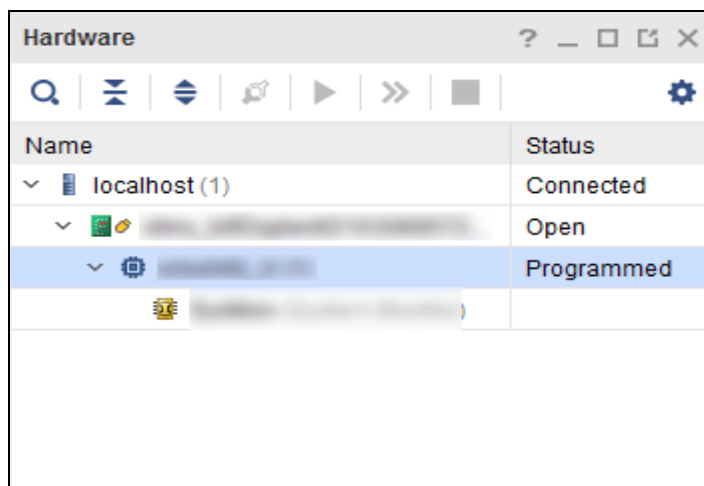
4-4-9. Right-click `xcku040_0` and select **Program Device.****Figure 6-18: Programming the Device**

The Program Device dialog box opens.

Observe that the bit file that will be used to program the device has been included in the Bitstream file field.

4-4-10. Click **Program.**

When programming the FPGA is complete, the status will show Programmed.

**Figure 6-19: Programming Status of the Device**

4-5. Launch GtkTerm and set the port configuration.

4-5-1. Click the **GtkTerm** icon from the quick launch toolbar (📁).

Alternatively, GtkTerm can be launched from a Linux terminal window (**Ctrl + Alt + T**) and entering:

```
[host] $ sudo gterm
```

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.

4-5-2. Select **Configuration > Port** to open the Configuration dialog box.

4-5-3. Identify the port associated with your board and set the port as **/dev/ttyUSBx** (where x could be 0, 1, 2, 3, etc.)

4-5-4. Set the baud rate to **115200**.

4-5-5. 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.

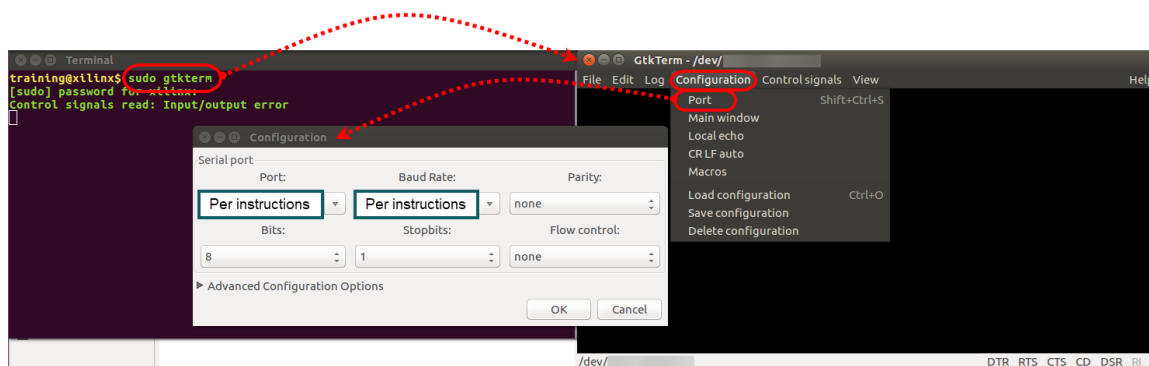


Figure 6-20: Opening GtkTerm and Selecting the Port Configuration

[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 with another name; however, you will then need to load the configuration each time you start GtkTerm.

4-5-6. Click **OK** to save the settings and leave the terminal open.

4-6. [Windows Users]: Configure the Tera Term terminal.

- 4-6-1. Select **All Programs > Tera Term > Tera Term** to open the Tera Term terminal.
- 4-6-2. Select the **Serial** option.
- 4-6-3. Select the appropriate COM port.

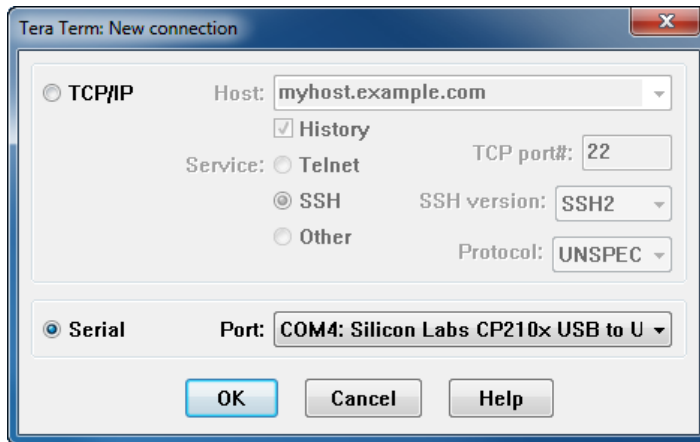


Figure 6-21: HyperTerminal Connection Description

Note: The COM port setting is specific to the computer being used and may need to be different than shown.

- 4-6-4. Click **OK**.
- 4-6-5. Select **Setup > Serial port** and select the following options.
 - Baud rate: **115200**
 - Data: **8 bit**
- 4-6-6. Click **OK**.
- 4-6-7. Select **Setup > Terminal**.
- 4-6-8. Select **Local echo** and click **OK**.

4-7. Verify that the design works on the evaluation board.

- 4-7-1. Type any character into the Tera Term window and observe the pattern of the LEDs on the board.

The pattern of the LEDs on the KCU105 board should reflect the ASCII equivalent of the character entered in Tera Term.

For a list of ASCII characters and their hexadecimal representation, see the *ascii_table.pdf* file in the `$sv_interfaces/support` directory.

Pressing the CPU_RESET button on the KCU105 board should reset the design.

- 4-7-2. Close GtkTerm/Tera Term.
- 4-7-3. Select **File > Close Hardware Manager** to close the hardware manager.
- 4-7-4. Power off the board.

Using Interfaces Across Hierarchy Levels

Step 5

In this step, you will validate that interfaces can be used across hierarchy levels.

5-1. Include interface in a lower-level module.

5-1-1. Determine which other module in the *uart_led* design can use the interface *ifc_signals*.

Hint: Descend into the *uart_rx* module.

5-1-2. Select the **Sources** window and expand **top > uart_led > uart_rx** in the Hierarchy tab.

5-1-3. Double-click the **uart_rx_ctl** module.

5-1-4. Include the interface in *uart_rx_ctl* in the same way that the interface (*ifc_signals*) was included in *uart_rx*.

5-1-5. Save the **uart_rx_ctl.sv** file.

5-2. Update the *uart_rx* module.

5-2-1. Start updating *uart_rx* module by first instantiating the interface, since *uart_rx* is the parent module for *uart_rx_ctl*.

Hint: Recall how the interface was instantiated at the *uart_led* module.

Note that in *uart_led*, the interface ports (*clk_rst* and *rst_clk_rx*) were connected to the *uart_led* ports with the same names.

However, in *uart_rx*, the same interface ports will be connected to signals that are internal to the modport used by *uart_rx*.

5-2-2. Use *assign* statements to explicitly connect the interface signals that are not declared as interface ports.

Example:

```
module uart_rx (
    // Write side inputs
    ifc_signals.uart_rx_signals uart_rx_ifc, // Interface used as ports

    input      rxd_i,          // RS232 RXD pin - Directly from pad
    output     rxd_clk_rx,     // RXD pin after synchronization to clk_rx
    output     frm_err         // The STOP bit was not detected
);

    ifc_signals uart_rx_top_ifc ( // Instantiating the interface
        .clk_rx (uart_rx_ifc.clk_rx),
        .rst_clk_rx (uart_rx_ifc.rst_clk_rx)
    );

    // Connecting rx_data and rx_data_rdy signals between uart_rx_ifc and uart_rx_top_ifc
    assign uart_rx_ifc.rx_data = uart_rx_top_ifc.rx_data;
    assign uart_rx_ifc.rx_data_rdy = uart_rx_top_ifc.rx_data_rdy;
```

Figure 6-22: Using assign Statements to Connect Interfaces Hierarchically

The *assign* statements are required because otherwise the interface signals *rx_data* and *rx_data_rdy* will remain unconnected.

In *uart_led*, the interface *uart_led_ifc* was used to connect both *uart_rx* and *led_ctl*.

However, in *uart_rx*, the interface *uart_rx_ifc* does not connect *uart_rx_ctl* to another module that uses the same interface. Hence, *rx_data* and *rx_data_rdy* would be left unconnected.

5-2-3. Connect the interfaces in the module instantiation of *uart_rx_ctl* as in *uart_led*.

5-2-4. Save the **uart_rx.sv** file.

5-3. Synthesize the design.

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

5-3-1. Resolve any reported errors.

5-4. Generate the bitstream for the design.

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

5-5. Program the Kintex FPGA on the KCU105 board.

5-5-1. Power on the board.

5-5-2. Select the **Generate Bitstream** option in the Implementation Completed dialog box after the implementation completes.

5-5-3. Click **OK**.

5-5-4. Select **Open Hardware Manager** in the Bitstream Generation Completed dialog box once the bitstream is generated.

5-5-5. Click **OK**.

The hardware manager will open in the workspace.

5-5-6. Click **Open target > Recent targets** in the hardware manager and select the target used in the previous section.

5-5-7. Follow the steps in the previous section to download and verify the design on the evaluation board.

5-6. Close GtkTerm/Tera Term.**5-7. Power off the board.****5-8. Close the Vivado Design Suite.**

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

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

5-9. [Optional] [Only for local VMs—not for CloudShare] Clean up the file system.

Using the GUI:

5-9-1. Using the graphical browser (Windows: press the <**Windows**> key + <**E**>; Linux: press <**Ctrl** + **N**>), navigate to `$TRAINING_PATH/sv_interfaces`.

5-9-2. Select **sv_interfaces**.

5-9-3. Press <**Delete**>.

-- OR --

Using the command line:

5-9-4. Open a terminal window (Windows: press the <**Windows**> key + <**R**>, then enter **cmd**; Linux: press <**Ctrl** + **Alt** + **T**>).

5-9-5. Enter the following command to delete the contents of the workspace:

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

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

Summary

Here you learned how to define and use SystemVerilog interfaces in an RTL design. You also learned how to download a design onto an evaluation board using the Vivado Design Suite.

Answers

Interface Module (ifc_signals)

```
`timescale 1ns / 1ps

interface ifc_signals(
    input clk_rx,
    input rst_clk_rx
);

// Signals from uart_rx to led_ctl

logic [7:0] rx_data;
logic rx_data_rdy;

modport uart_rx_signals(
    input clk_rx,
    input rst_clk_rx,

    output rx_data,
    output rx_data_rdy
);

modport led_ctl_signals(
    input clk_rx,
    input rst_clk_rx,

    input rx_data,
    input rx_data_rdy
);

endinterface
```

uart_rx (when the interface is not used hierarchically i.e., when uart_rx_ctl does not have an interface):

```
`timescale 1ns/1ps

module uart_rx (
    // Write side inputs

    ifc_signals.uart_rx_signals uart_rx_ifc,

    /*  input          clk_rx,          // Clock input
       output          rx_data_rdy,    // Ready signal for rx_data
       output          [7:0] rx_data,
    input          rst_clk_rx,        */

    input          rxd_i,             // RS232 RXD pin - Directly from pad
    output          rxd_clk_rx,       // RXD pin after synchronization to clk_rx
```

```

    output          frm_err          // The STOP bit was not detected
);

//*****
// Parameter definitions
//*****

    parameter BAUD_RATE    = 115_200;          // Baud rate
    parameter CLOCK_RATE   = 50_000_000;

//*****
// Reg declarations
//*****

//*****
// Wire declarations
//*****

    wire          baud_x16_en;  // 1-in-N enable for uart_rx_ctl FFs

//*****
// Code
//*****

/* Synchronize the RXD pin to the clk_rx clock domain. Since RXD changes
 * very slowly wrt. the sampling clock, a simple metastability hardener is
 * sufficient */

meta_harden meta_harden_rxd_i0 (
    .clk_dst      (uart_rx_ifc.clk_rx),
    .rst_dst      (uart_rx_ifc.rst_clk_rx),
    .signal_src   (rxd_i),
    .signal_dst   (rxd_clk_rx)
);

uart_baud_gen #
( .BAUD_RATE  (BAUD_RATE),
  .CLOCK_RATE (CLOCK_RATE)
) uart_baud_gen_rx_i0 (
    .clk      (uart_rx_ifc.clk_rx),
    .rst      (uart_rx_ifc.rst_clk_rx),
    .baud_x16_en (baud_x16_en)
);

uart_rx_ctl uart_rx_ctl_i0 (
    .clk_rx      (uart_rx_ifc.clk_rx),
    .rst_clk_rx  (uart_rx_ifc.rst_clk_rx),
    .baud_x16_en (baud_x16_en),
    .rxd_clk_rx  (rxd_clk_rx),

```



```

        .rx_data_rdy (uart_rx_ifc.rx_data_rdy),
        .rx_data      (uart_rx_ifc.rx_data),
        .frm_err      (frm_err)
    );
endmodule

```

led_ctl

```

`timescale 1ns/1ps

module led_ctl (
    // Write side inputs
    ifc_signals.led_ctl_signals led_ctl_ifc,
    output logic [7:0] led_o      // The LED outputs
);

//*****
// Logic declarations
//*****

    logic          old_rx_data_rdy;
    logic [7:0]    char_data;

//*****
// Code
//*****

    always_ff @(posedge led_ctl_ifc.clk_rx)
    begin
        if (led_ctl_ifc.rst_clk_rx)
            begin
                old_rx_data_rdy <= 1'b0;
                char_data        <= 8'b0;
                led_o            <= 8'b0;
            end
        else
            begin
                // Capture the value of rx_data_rdy for edge detection
                old_rx_data_rdy <= led_ctl_ifc.rx_data_rdy;

                // If rising edge of rx_data_rdy, capture rx_data
                if (led_ctl_ifc.rx_data_rdy && !old_rx_data_rdy)
                    begin
                        char_data <= led_ctl_ifc.rx_data;
                    end
            end
        end
    end

```

```

        led_o <= char_data;
    end // if !rst
    end // always_ff
endmodule

```

uart_led

```

`timescale 1ns/1ps
module uart_led (
    // Write side inputs
    input          clk_rx,          // Clock input
    input          rst_clk_rx,      // Active HIGH reset
    input          rxd_i,           // RS232 RXD pin
    output [7:0] led_o              // 8 LED outputs
);

ifc_signals uart_led_ifc (
    .clk_rx (clk_rx),
    .rst_clk_rx (rst_clk_rx)
);

//*****
// Parameter definitions
//*****

parameter BAUD_RATE          = 115_200;
parameter CLOCK_RATE         = 125_000_000;

uart_rx #(
    .CLOCK_RATE    (CLOCK_RATE),
    .BAUD_RATE     (BAUD_RATE)
) uart_rx_i0 (
    .uart_rx_ifc (uart_led_ifc),
    .rxd_i       (rxd_i),
    .rxd_clk_rx  (),
    .frm_err     ()
);

led_ctl led_ctl_i0 (
    .led_ctl_ifc (uart_led_ifc),
    .led_o       (led_o)
);

endmodule

```

uart_rx_ctl (when the interface is used hierarchically; i.e., when uart_rx_ctl uses an interface)

```

`timescale 1ns/1ps

module uart_rx_ctl (
    // Write side inputs
    ifc_signals.uart_rx_signals uart_rx_ctl_ifc,
    input          baud_x16_en,    // 16x oversampling enable
    input          rxd_clk_rx,     // RS232 RXD pin - after sync to clk_rx
    output logic    frm_err        // The STOP bit was not detected
);

//*****
// Parameter definitions
//*****

// State encoding for main FSM
enum {IDLE, START, DATA, STOP} state;

//*****
// Logic declarations
//*****

logic [3:0]    over_sample_cnt;    // Oversample counter - 16 per bit
logic [2:0]    bit_cnt;            // Bit counter - which bit are we RXing

//*****
// Wire declarations
//*****

wire          over_sample_cnt_done; // We are in the middle of a bit
wire          bit_cnt_done;         // This is the last data bit

//*****
// Code
//*****

// Main state machine
always_ff @(posedge uart_rx_ctl_ifc.clk_rx)
begin
    if (uart_rx_ctl_ifc.rst_clk_rx)
        begin
            state    <= IDLE;
        end
    else
        begin
            if (baud_x16_en)

```

```
begin
  case (state)
    IDLE: begin
      // On detection of rxd_clk_rx being low, transition to the START
      // state
      if (!rxd_clk_rx)
        begin
          state <= START;
        end
      end // IDLE state
    START: begin
      // After 1/2 bit period, re-confirm the start state
      if (over_sample_cnt_done)
        begin
          if (!rxd_clk_rx)
            begin
              // Was a legitimate start bit (not a glitch)
              state <= DATA;
            end
          else
            begin
              // Was a glitch - reject
              state <= IDLE;
            end
          end // if over_sample_cnt_done
        end // START state
      DATA: begin
        // Once the last bit has been received, check for the stop bit
        if (over_sample_cnt_done && bit_cnt_done)
          begin
            state <= STOP;
          end
        end // DATA state
      STOP: begin
        // Return to idle
        if (over_sample_cnt_done)
          begin
```

```
        state <= IDLE;
    end
    end // STOP state
endcase
end // if baud_x16_en
end // if rst_clk_rx
end // always_ff
// Oversample counter
// Pre-load to 7 when a start condition is detected (rx_clk_rx is 0 while in
// IDLE) - this will get us to the middle of the first bit.
// Pre-load to 15 after the START is confirmed and between all data bits.
always_ff @(posedge uart_rx_ctl_ifc.clk_rx)
begin
    if (uart_rx_ctl_ifc.rst_clk_rx)
    begin
        over_sample_cnt    <= 4'd0;
    end
    else
    begin
        if (baud_x16_en)
        begin
            if (!over_sample_cnt_done)
            begin
                over_sample_cnt <= over_sample_cnt - 1'b1;
            end
            else
            begin
                if ((state == IDLE) && !rx_clk_rx)
                begin
                    over_sample_cnt <= 4'd7;
                end
                else if ( ((state == START) && !rx_clk_rx) || (state == DATA) )
                begin
                    over_sample_cnt <= 4'd15;
                end
            end
        end
    end
end
```

```
        end // if baud_x16_en
    end // if rst_clk_rx
end // always_ff
assign over_sample_cnt_done = (over_sample_cnt == 4'd0);
// Track which bit we are about to receive
// Set to 0 when we confirm the start condition
// Increment in all DATA states
always_ff @(posedge uart_rx_ctl_ifc.clk_rx)
begin
    if (uart_rx_ctl_ifc.rst_clk_rx)
        begin
            bit_cnt    <= 3'b0;
        end
    else
        begin
            if (baud_x16_en)
                begin
                    if (over_sample_cnt_done)
                        begin
                            if (state == START)
                                begin
                                    bit_cnt <= 3'd0;
                                end
                            else if (state == DATA)
                                begin
                                    bit_cnt <= bit_cnt + 1'b1;
                                end
                        end
                    end // if over_sample_cnt_done
                end // if baud_x16_en
            end // if rst_clk_rx
        end // always_ff
    assign bit_cnt_done = (bit_cnt == 3'd7);
    // Capture the data and generate the rdy signal
    // The rdy signal will be generated as soon as the last bit of data
    // is captured - even though the STOP bit hasn't been confirmed. It will
    // remain asserted for one BIT period (16 baud_x16_en periods)
```

```
always_ff @(posedge uart_rx_ctl_ifc.clk_rx)
begin
  if (uart_rx_ctl_ifc.rst_clk_rx)
    begin
      uart_rx_ctl_ifc.rx_data      <= 8'b0000_0000;
      uart_rx_ctl_ifc.rx_data_rdy <= 1'b0;
    end
  else
    begin
      if (baud_x16_en && over_sample_cnt_done)
        begin
          if (state == DATA)
            begin
              uart_rx_ctl_ifc.rx_data[bit_cnt] <= rxd_clk_rx;
              uart_rx_ctl_ifc.rx_data_rdy      <= (bit_cnt == 3'd7);
            end
          else
            begin
              uart_rx_ctl_ifc.rx_data_rdy      <= 1'b0;
            end
          end
        end
      end // if rst_clk_rx
    end // always_ff

  // Framing error generation
  // Generate for one baud_x16_en period as soon as the framing bit
  // is supposed to be sampled

  always_ff @(posedge uart_rx_ctl_ifc.clk_rx)
  begin
    if (uart_rx_ctl_ifc.rst_clk_rx)
      begin
        frm_err      <= 1'b0;
      end
    else
      begin
        if (baud_x16_en)
          begin
            if ((state == STOP) && over_sample_cnt_done && !rxd_clk_rx)
```

```

        begin
            frm_err <= 1'b1;
        end

        else
            begin
                frm_err <= 1'b0;
            end
        end // if baud_x16_en
    end // if rst_clk_rx
end // always_ff
endmodule

```

uart_rx (when the interface is used hierarchically i.e., when uart_rx_ctl uses an interface)

```

`timescale 1ns/1ps

module uart_rx (
    // Write side inputs
    ifc_signals.uart_rx_signals uart_rx_ifc, // Interface used as ports
    input          rxd_i,                    // RS232 RXD pin - Directly from pad
    output         rxd_clk_rx,               // RXD pin after synchronization to clk_rx
    output         frm_err                   // The STOP bit was not detected
);

    ifc_signals uart_rx_top_ifc ( // Instantiating the interface
        .clk_rx (uart_rx_ifc.clk_rx),
        .rst_clk_rx (uart_rx_ifc.rst_clk_rx)
    );

    // Connecting rx_data and rx_data_rdy signals between uart_rx_ifc and uart_rx_top_ifc
    assign uart_rx_ifc.rx_data = uart_rx_top_ifc.rx_data;
    assign uart_rx_ifc.rx_data_rdy = uart_rx_top_ifc.rx_data_rdy;

    //*****
    // Parameter definitions
    //*****

    parameter BAUD_RATE    = 115_200;           // Baud rate
    parameter CLOCK_RATE   = 50_000_000;

    //*****
    // Wire declarations
    //*****

    wire          baud_x16_en; // 1-in-N enable for uart_rx_ctl FFs

```



```
//*****  
// Code  
//*****  
  
/* Synchronize the RXD pin to the clk_rx clock domain. Since RXD changes  
* very slowly wrt. the sampling clock, a simple metastability hardener is  
* sufficient */  
  
meta_harden meta_harden_rxd_i0 (  
    .clk_dst      (uart_rx_ifc.clk_rx),  
    .rst_dst      (uart_rx_ifc.rst_clk_rx),  
    .signal_src   (rxd_i),  
    .signal_dst   (rxd_clk_rx)  
);  
  
uart_baud_gen #  
( .BAUD_RATE  (BAUD_RATE),  
  .CLOCK_RATE (CLOCK_RATE)  
) uart_baud_gen_rx_i0 (  
    .clk      (uart_rx_ifc.clk_rx),  
    .rst      (uart_rx_ifc.rst_clk_rx),  
    .baud_x16_en (baud_x16_en)  
);  
  
uart_rx_ctl uart_rx_ctl_i0 (  
    .uart_rx_ctl_ifc (uart_rx_top_ifc),  
    .baud_x16_en (baud_x16_en),  
    .rxd_clk_rx  (rxd_clk_rx),  
    .frm_err     (frm_err)  
);  
  
endmodule
```