Designing a Simple Process

2021.1

Abstract

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

Objectives

After completing this lab, you will be able to:

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

Introduction

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

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

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

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

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

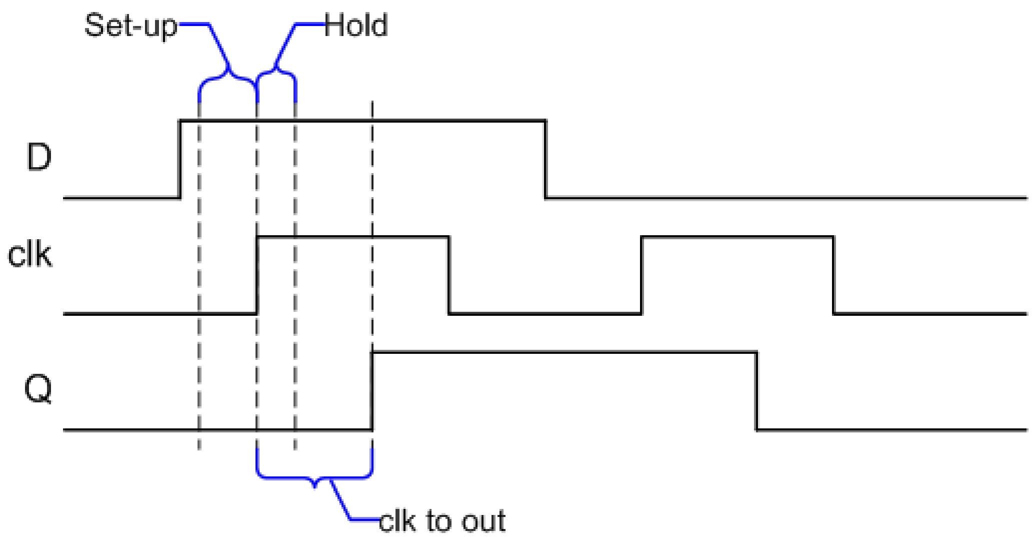


Figure 3-1: Valid Flip-Flop Timing

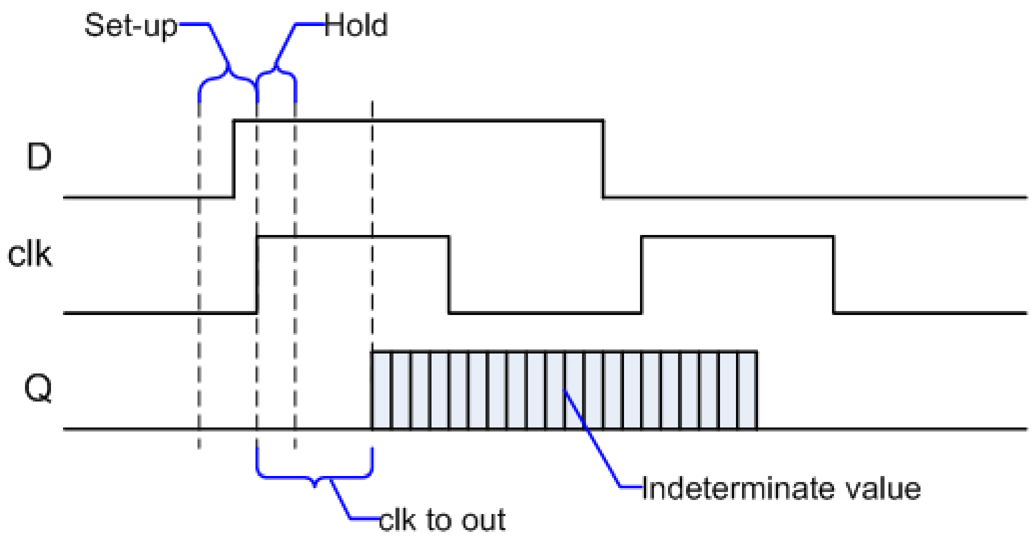


Figure 3-2: Illegal (Bad) Setup Time and Its Consequences

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

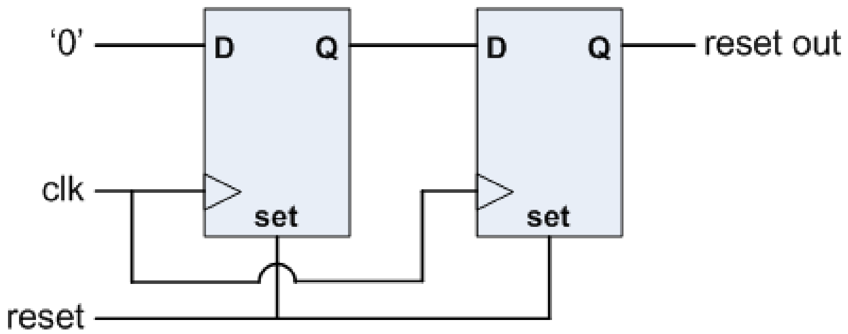


Figure 3-3: Circuit for This Lab

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

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

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

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

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

Understanding the Lab Environment

The labs and demos provided in this course are designed to run 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:

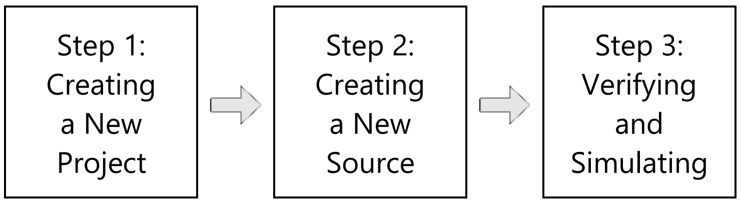
Environment Variable Name	**Description**
`$TRAINING_PATH`	Points to the space allocated for students to work through the labs. This directory includes prebuilt images and starting points for the labs and demos. The customer training VM sets `$TRAINING_PATH` to the `/home/xilinx/training` directory.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

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


Step 1: Creating a New Project
Step 2: Creating a New Source
Step 3: Verifying and Simulating

Creating a New Project Step 1

1-1. Launch the Vivado Design Suite.

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

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

Browse to the `$TRAINING_PATH/processes/lab/KCU105` directory.

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

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

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

Creating a New VHDL Source Step 2

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

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

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

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

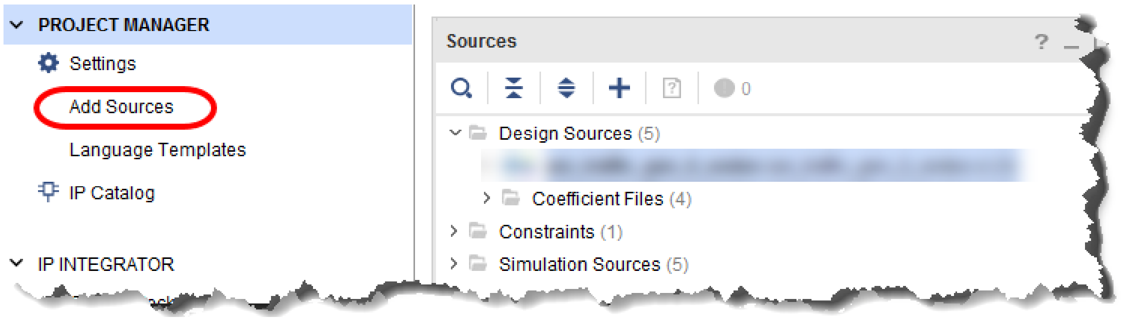
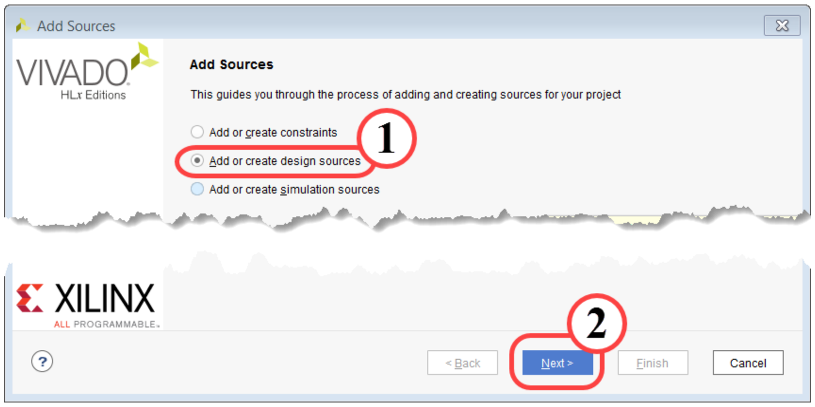


Figure 3-4: Selecting Add Sources

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



Add Sources

This guides you through the process of adding and creating sources for your project

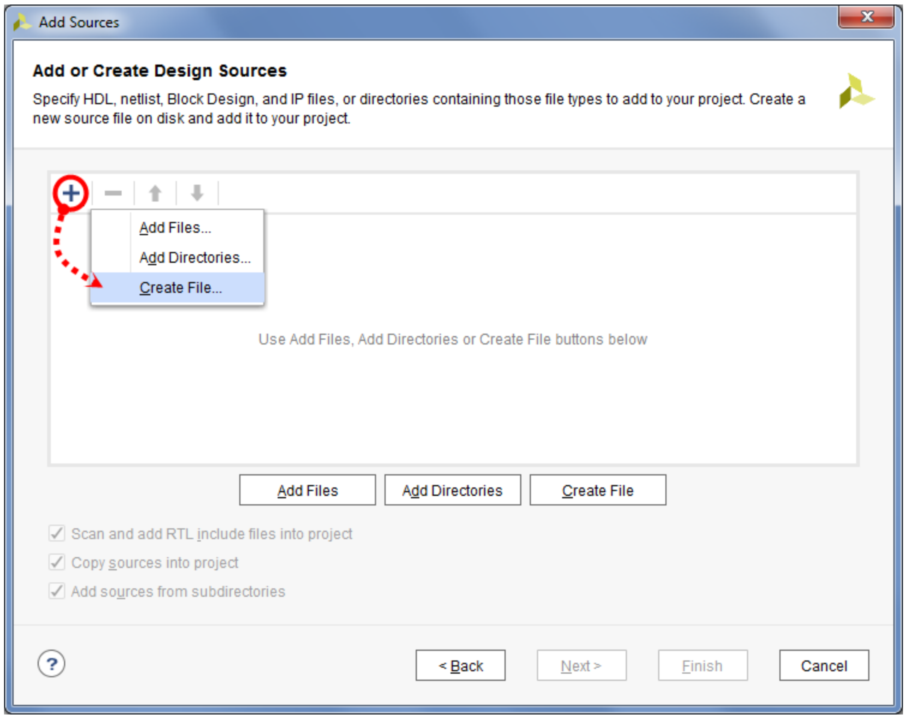
Add or create constraints
[✓ Add or create design sources]
Add or create simulation sources

Figure 3-5: 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**.



Add or Create Design Sources

Specify HDL, netlist, Block Design, and IP files, or directories containing those file types to add to your project. Create a new source file on disk and add it to your project.

Use Add Files, Add Directories or Create File buttons below

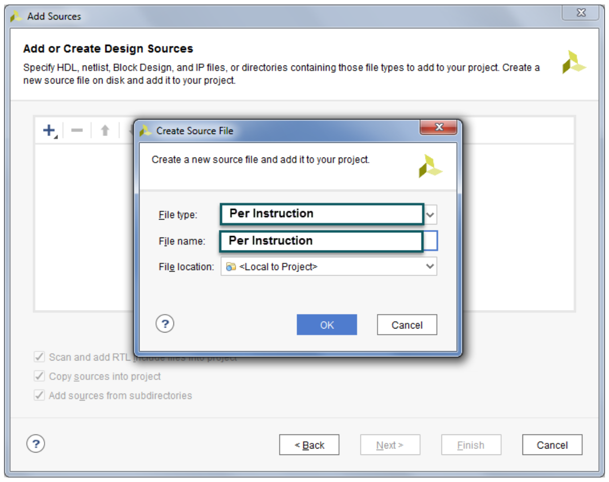
✓ Scan and add RTL include files into project
✓ Copy sources into project
✓ Add sources from subdirectories

Figure 3-6: Selecting Create File

The Create Source File dialog box opens.

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

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


Add Sources

Add or Create Design Sources

Specify HDL, netlist, Block Design, and IP files, or directories containing those file types to add to your project. Create a new source file on disk and add it to your project.

Create Source File

Create a new source file and add it to your project.

File type: Per Instruction

File name: Per Instruction

File location: <Local to Project>

OK
Cancel
Scan and add RTL include files into project
Copy sources into project
Add sources from subdirectories
< Back
Next >
Finish
Cancel

Figure 3-7: Entering File Name and Type

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

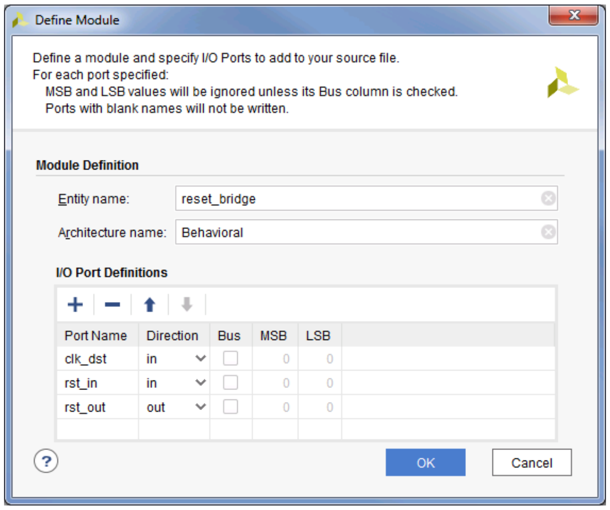
2-1-8. Click **Finish** to add the new source file(s).

The Define Module dialog box opens.

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

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

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


Define Module

Define a module and specify I/O Ports to add to your source file.
For each port specified:
MSB and LSB values will be ignored unless its Bus column is checked.
Ports with blank names will not be written.

Module Definition

Entity name: reset_bridge

Architecture name: Behavioral

I/O Port Definitions

Port Name	Direction	Bus	MSB	LSB
clk_dst	in	☐	0	0
rst_in	in	☐	0	0
rst_out	out	☐	0	0

Figure 3-8: Defining Ports for reset_bridge

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

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

Double-click **reset_bridge.vhd**.

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

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

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

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

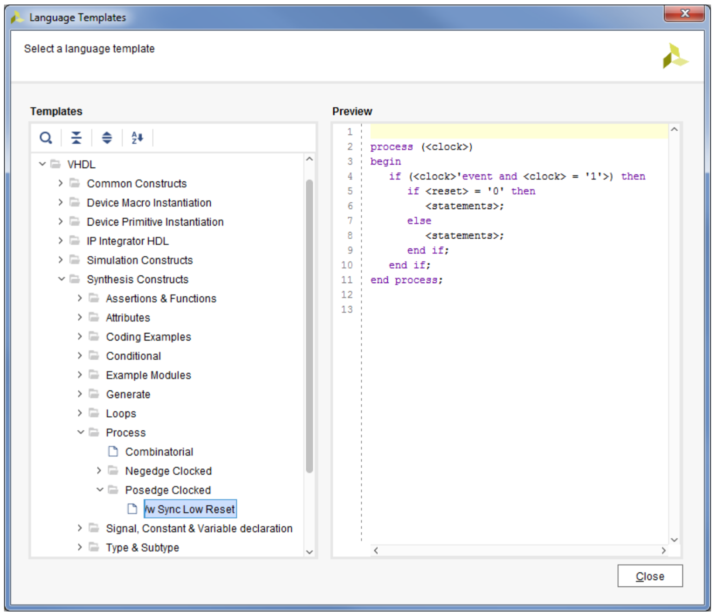
Using the Language Templates, select the snippet for a process clocked on a Posedge and with an asynchronous reset.

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

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

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

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



```

process (<clock>)
begin
   if (<clock>'event and <clock> = '1'>) then
      if <reset> = '0' then
         <statements>;
      else
         <statements>;
      end if;
   end if;
end process;


```

Figure 3-9: Partially Expanded Language Template List

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

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

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

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

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

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

The asterisk means that this file is unsaved.

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

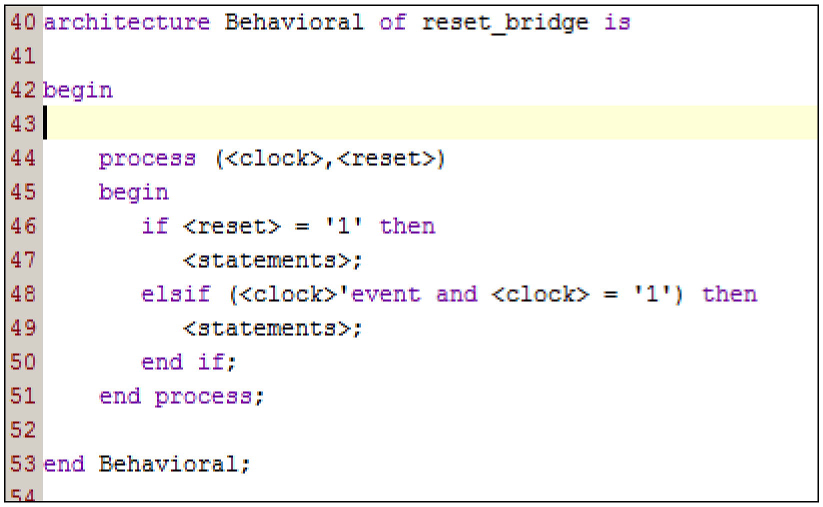
2-5-5. Right-click and select **Paste**.

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

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

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

Your code should look like the figure shown below.



```
architecture Behavioral of reset_bridge is

begin

    process (<clock>,<reset>)
    begin
       if <reset> = '1' then
          <statements>;
       elsif (<clock>'event and <clock> = '1') then
          <statements>;
       end if;
    end process;

end Behavioral;
```

Figure 3-10: Code Changed to Asynchronous High Reset

2-6. The snippet represents generic code.

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

Take your time and examine the code carefully.

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

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

2-6-1. Make the following substitutions:

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

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

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

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

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

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

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

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

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

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

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

Question 1

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

The design would have only had one register.

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

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

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

Verifying with the Viewer and Simulating Step 3

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

3-1. Run synthesis.

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

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

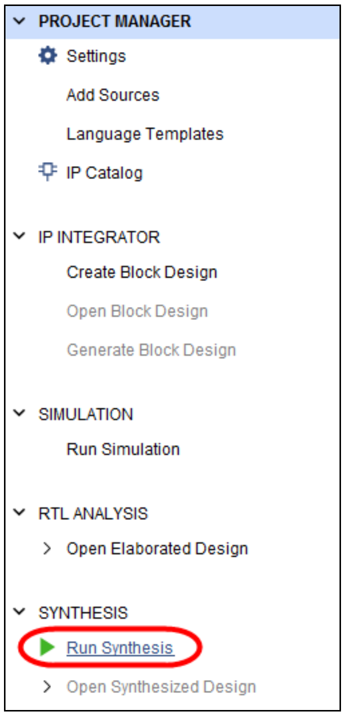
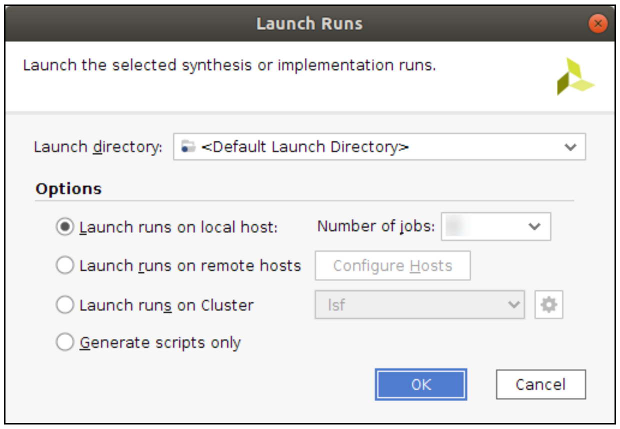


Figure 3-11: Selecting Run Synthesis

The Launch Runs dialog box opens.


Launch Runs

Launch the selected synthesis or implementation runs.

Launch directory: <Default Launch Directory>

Options

- (●) Launch runs on local host: Number of jobs:
- ( ) Launch runs on remote hosts [Configure Hosts]
- ( ) Launch runs on Cluster lsf
- ( ) Generate scripts only

[OK] [Cancel]

Figure 3-12: Setting the Launch Run Configuration

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

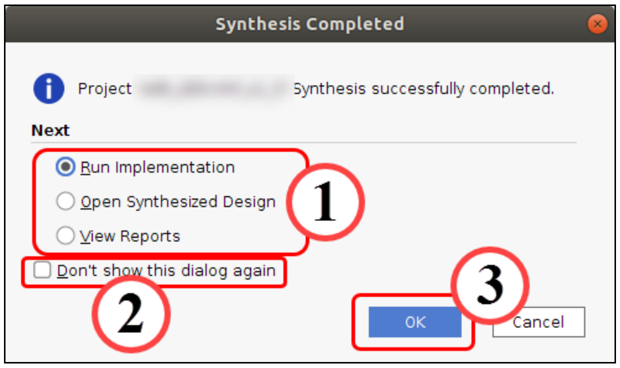
3-1-2. Click **OK** to launch the runs.

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

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

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

3-1-5. [Optional] If you are familiar with accessing these various capabilities, you can disable this dialog box from appearing again by selecting **Don't show this dialog again** (2).


Synthesis Completed

Project Synthesis successfully completed.

Next

- (●) Run Implementation
- ( ) Open Synthesized Design
- ( ) View Reports

☐ Don't show this dialog again

[OK] [Cancel]

Figure 3-13: Selecting After Synthesis Options

3-1-6. Click **OK** to take the action you just selected (3) or click **Cancel** to simply close the dialog box.

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

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

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

3-2-1. Open the synthesized design if it is not already opened

3-2-2. Right-click **reset_bridge** in the Netlist view and select **Schematic**.

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

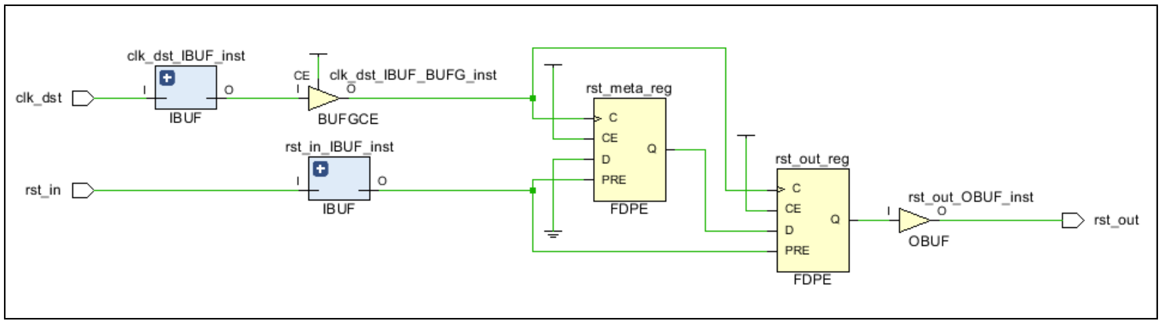


Figure 3-14: Final Design (KCU105)

Question 2

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

The synthesized design does not look like this, but my elaborated design does. I think this is because the rst_meta is a variable. I do not know how this design shows it after being synthesized

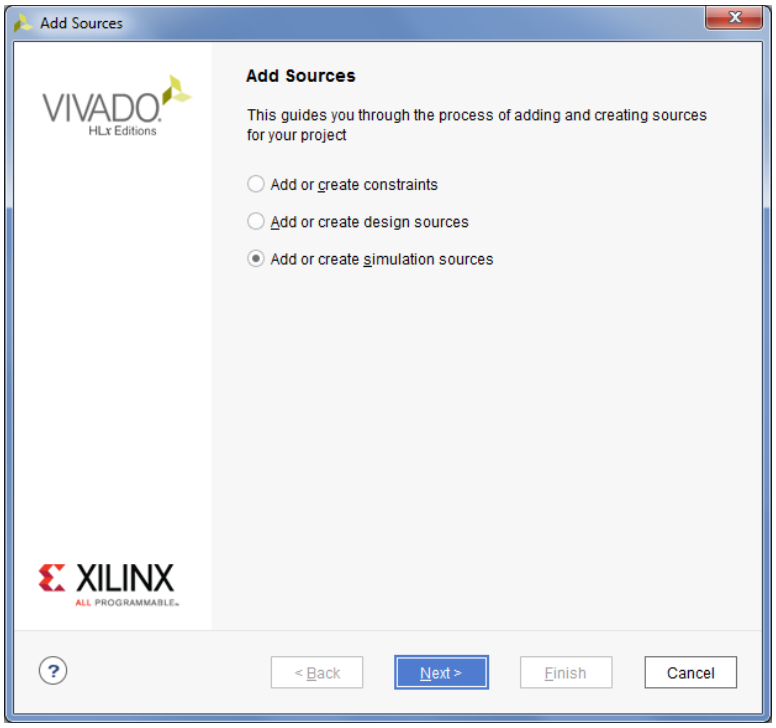
3-2-3. Close the synthesized design.

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

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

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

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


Add Sources

Add Sources

This guides you through the process of adding and creating sources for your project

Add or create constraints [✓ Add or create simulation sources] Add or create design sources

Figure 3-15: Add Sources for Simulation

The Add or Create Simulation Sources dialog box opens.

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

3-3-4. Select **reset_bridge_tb.vhd** from the location `$TRAINING_PATH/processes/support` and click **OK**.

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

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

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

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

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

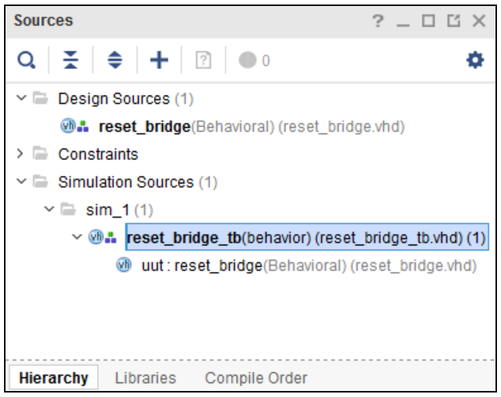

Sources
Design Sources (1)
reset_bridge(Behavioral) (reset_bridge.vhd)
Constraints
Simulation Sources (1)
sim_1 (1)
reset_bridge_tb(behavior) (reset_bridge_tb.vhd) (1)
uut : reset_bridge(Behavioral) (reset_bridge.vhd)
Hierarchy
Libraries
Compile Order

Figure 3-16: Opening the Testbench

Note how the various signal stimuli are generated.

3-5. Launch the simulation for reset_bridge_tb.

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

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

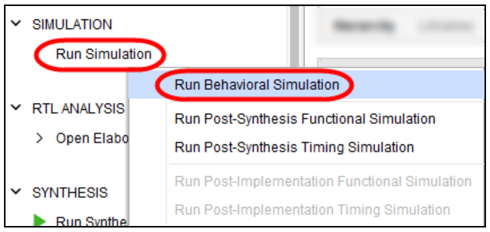

SIMULATION
Run Simulation
Run Behavioral Simulation
RTL ANALYSIS
Open Elabo
Run Post-Synthesis Functional Simulation
Run Post-Synthesis Timing Simulation
SYNTHESIS
Run Post-Implementation Functional Simulation
Run Post-Implementation Timing Simulation
Run Synthe

Figure 3-17: Launching the Vivado Simulator Tool

This will launch the Vivado Simulator.

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

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

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

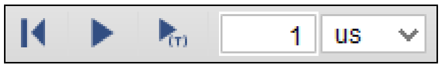


Figure 3-18: Vivado Simulator Control Toolbar

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

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

Question 3

Is the design working?

Yes, it is working as intended. There's a delay before the output registers the data

Question 4

What happens when the reset and clock are asserted simultaneously?

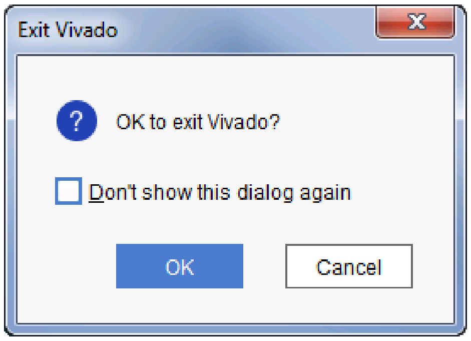
It sets rst_out immediately.

3-7. Close the simulation.

3-8. Close the Vivado Design Suite.

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

The Exit Vivado dialog box opens.


Exit Vivado

OK to exit Vivado?

Don't show this dialog again

Figure 3-19: Exit Vivado Dialog Box

3-8-2. 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/processes` directory.

3-9. [Optional] [Only for local VMs—not for CloudShare] Clean up the file system.

Using the GUI:

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

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

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

-- OR --

Using the command line:

3-9-4. Open a terminal window (Windows: press the <**Windows**> key + <**R**>, then enter **cmd**; Linux: press <**Ctrl** + **Alt** + **T**>).

3-9-5. Enter the following command to delete the contents of the workspace:

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

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

Summary

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

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

Answers

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

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

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

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

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

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

3. Is the design working?

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

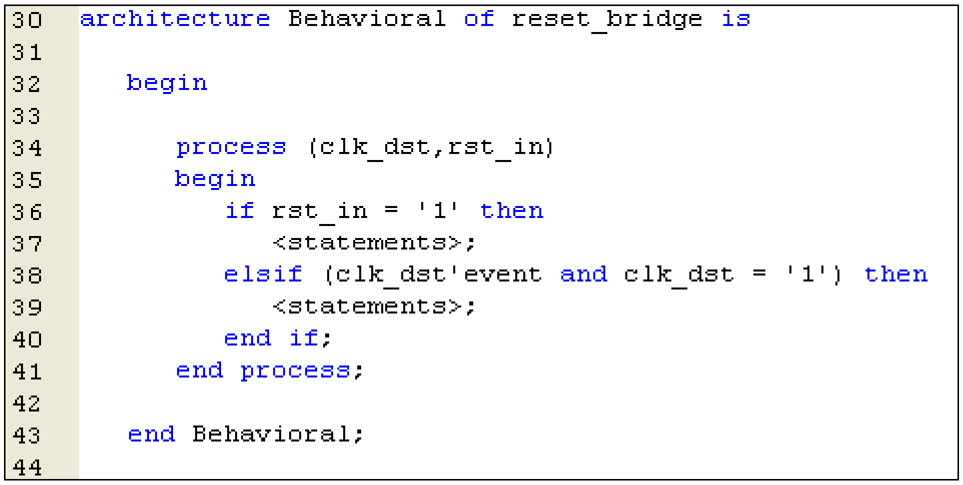
4. What happens when the reset and clock are asserted simultaneously?

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

Coding the Process

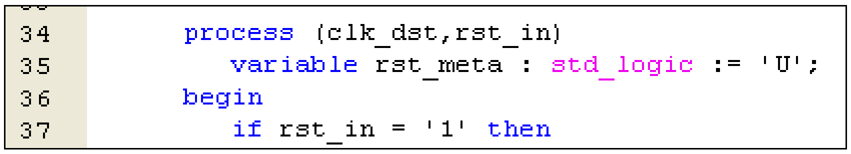
Making the following substitutions:

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



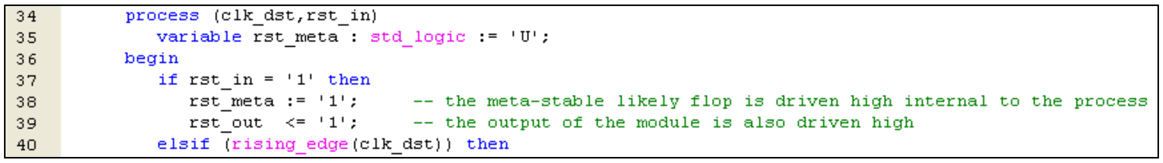
```
architecture Behavioral of reset_bridge is

   begin

      process (clk_dst,rst_in)
      begin
         if rst_in = '1' then
            <statements>;
         elsif (clk_dst'event and clk_dst = '1') then
            <statements>;
         end if;
      end process;

   end Behavioral;

```

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



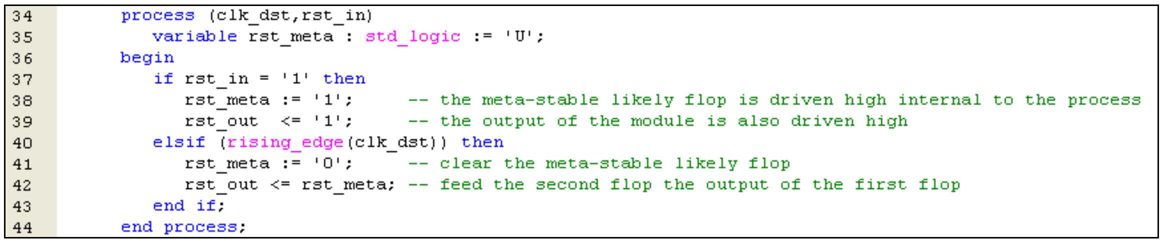
```
      process (clk_dst,rst_in)
         variable rst_meta : std_logic := 'U';
      begin
         if rst_in = '1' then
```

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



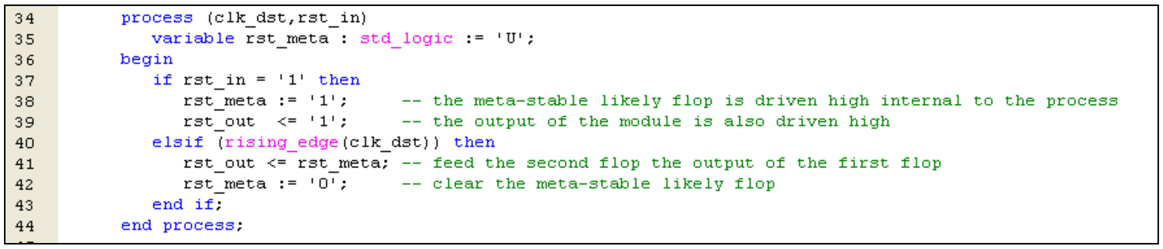
```
      process (clk_dst,rst_in)
         variable rst_meta : std_logic := 'U';
      begin
         if rst_in = '1' then
            rst_meta := '1';     -- the meta-stable likely flop is driven high internal to the process
            rst_out  <= '1';     -- the output of the module is also driven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

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



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

Placing the assignment to rst_meta AFTER that of rst_out:



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