See all versions of this document

Vivado Design Suite Tutorial

Logic Simulation

UG937 (v2021.1) July 14, 2021

XILINX®

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
07/14/2021 Version 2021.1	
General updates	Updated for Vivado® ML Editions 2021.1

Table of Contents

XILINX

Chapter 1

Vivado Simulator Overview

IMPORTANT! *This tutorial requires the use of the Kintex®-7 family of devices or UltraScale™ devices. If you do not have this device family installed, you must update your Vivado® tools installation. Refer to the Vivado Design Suite User Guide: Release Notes, Installation, and Licensing (UG973) for more information on Adding Design Tools or Devices to your installation.*

Introduction

This Xilinx® Vivado® Design Suite tutorial provides designers with an in-depth introduction to the Vivado simulator.

VIDEO: *You can also learn more about the Vivado simulator by viewing the quick take video at Vivado Logic Simulation.*

TRAINING: *Xilinx provides training courses that can help you learn more about the concepts presented in this document. Use these links to explore related courses:*

- Designing FPGAs Using the Vivado Design Suite 1 Training Course
- Designing FPGAs Using the Vivado Design Suite 2 Training Course
- Designing FPGAs Using the Vivado Design Suite 3 Training Course

The Vivado simulator is a Hardware Description Language (HDL) simulator that lets you perform behavioral, functional, and timing simulations for VHDL, Verilog, and mixed-language designs. The Vivado simulator environment includes the following key elements:

- **xvhdl and xvlog:** Parsers for VHDL and Verilog files, respectively, that store the parsed files into an HDL library on disk.
- **xelab:** HDL elaborator and linker command. For a given top-level unit, xelab loads up all sub-design units, translates the design units into executable code, and links the generated executable code with the simulation kernel to create an executable simulation snapshot.
- **xsim:** Vivado simulation command that loads a simulation snapshot to effect a batch mode simulation, or a GUI or Tcl-based interactive simulation environment.
- **Vivado Integrated Design Environment (IDE):** An interactive design-editing environment that provides the simulator user-interface.

Tutorial Description

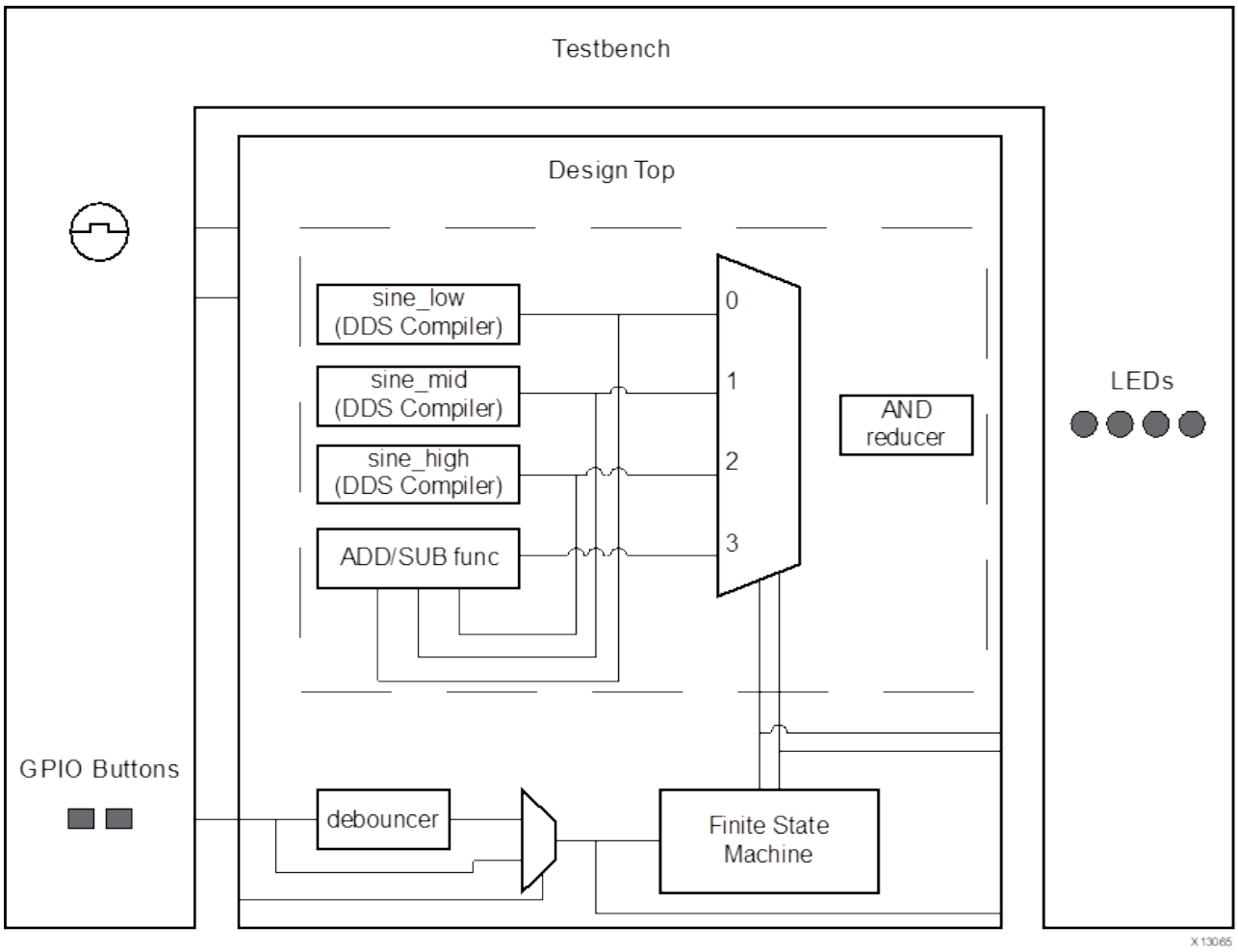
This tutorial demonstrates a design flow in which you can use the Vivado® simulator for performing behavioral, functional, or timing simulation from the Vivado Integrated Design Environment (IDE).

IMPORTANT! *Tutorial files are configured to run the Vivado simulator in a Windows environment. To run elements of this tutorial under the Linux operating system, some file modifications may be necessary.*

You can run the Vivado simulator in both Project Mode (using a Vivado design project to manage design sources and the design flow) and in Non-Project mode (managing the design more directly). For more information about Project Mode and Non-Project Mode, refer to the *Vivado Design Suite User Guide: Design Flows Overview* (UG892).

The following figure shows a block diagram of the tutorial design.

Figure 1: **Tutorial Design**



The tutorial design consists of the following blocks:

- A sine wave generator that generates high, medium, and low frequency sine waves; plus an amplitude sine wave (`sinegen.vhd`).
- DDS compilers that generate low, middle, and high frequency waves: (`sine_low.vhd`, `sine_mid.vhd`, and `sine_high.vhd`).
- A Finite State Machine (FSM) to select one of the four sine waves (`fsm.vhd`).
- A debouncer that enables switch-selection between the raw and the debounced version of the sine wave selector (`debounce.vhd`).
- A design top module that resets FSM and the sine wave generator, and then multiplexes the sine select results to the LED output (`sinegen_demo.vhd`).
- A simple testbench (`testbench.v`) to initiate the sine wave generator design that:
  - Generates a 200 MHz input clock for the design system clock, `sys_clk_p`.
  - Generates GPIO button selections.
  - Controls raw and debounced sine wave select.

Note: For more information about testbenches, see *Writing Efficient Test Benches* (XAPP199).

Locating Tutorial Design Files

There are separate project files and sources for each of the labs in this tutorial. You can find these at the link provided below or under **Support → Documentation → Development Tools (Product Type) → Hardware Development (Product Category) → Vivado – ML Editions (Product) → Tutorials (Doc Type)** on the Xilinx website.

1. Download the reference design files.
2. Extract the zip file contents into any write-accessible location.

   This tutorial refers to the extracted file contents of `ug937-design-files` directory as `<Extract_Dir>`.

RECOMMENDED: *You modify the tutorial design data while working through this tutorial. Use a new copy of the design files each time you start this tutorial.*

The following table describes the contents of the `ug937-design-files.zip` file.

Table 1: **Design File Contents**

Directories/Files	Description
`/completed`	Contains the completed files, and a Vivado 2021.x project of the tutorial design for reference. (x denotes the latest version of Vivado 2021 IDE)
`/scripts`	Contains the scripts you run during the tutorial.
`/sim`	Contains the `testbench.v` file.
`/sources`	Contains the HDL files necessary for the functional simulation.
`readme.txt`	`readme.txt` is a readme file about the contents and version history of this tutorial design.
`/uvm`	UVM example needed for Lab 5

Software and Hardware Requirements

This tutorial requires that the 2021.1 Vivado® ML Editions software release is installed. The following partial list describes the operating systems that the Vivado® ML Editions supports on x86 and x86-64 processor architectures:

- **Microsoft Windows Support:**
  - Windows 10 Professional (32-bit and 64-bit), English/Japanese
- **Linux Support:**
  - Red Hat Enterprise Workstation/Server 7.8-8.3 (64-bit), English/Japanese
  - SUSE Linux Enterprise 15.2 (32-bit and 64-bit)
  - CentOS 7.4 - 8.3 (64-bit), English/Japanese
  - Ubuntu Linux 16.04.5 LTS;16.04.6 LTS; 18.04.1 LTS; 18.04.2 LTS, 18.04.3 LTS; 18.04.4 LTS; and 18.04.5 LTS (64-bit), English/Japanese

Refer to the *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973) for a complete list and description of the system and software requirements.

XILINX

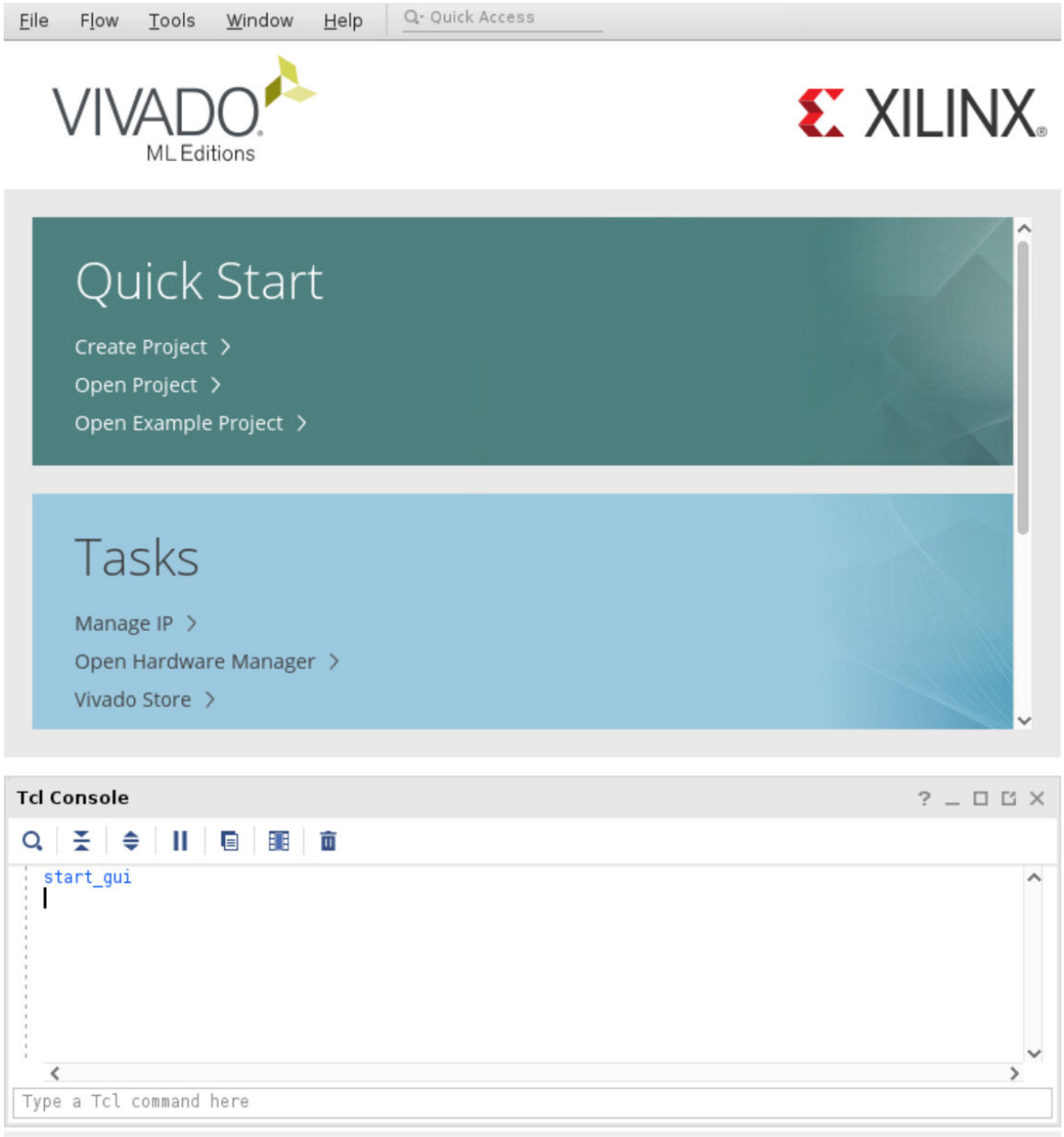
Chapter 2

Lab 1: Running the Simulator in Vivado IDE

In this lab, you create a new Vivado® Design Suite project, add HDL design sources, add IP from the Xilinx® IP catalog, and generate IP outputs needed for simulation. Then you run a behavioral simulation on an elaborated RTL design.

Step 1: Creating a New Project

The Vivado® Integrated Design Environment (IDE), as shown in the following figure, lets you launch simulation from within design projects, automatically generating the necessary simulation commands and files.



Create a new project for managing source files, add IP to the design, and run behavioral simulation:

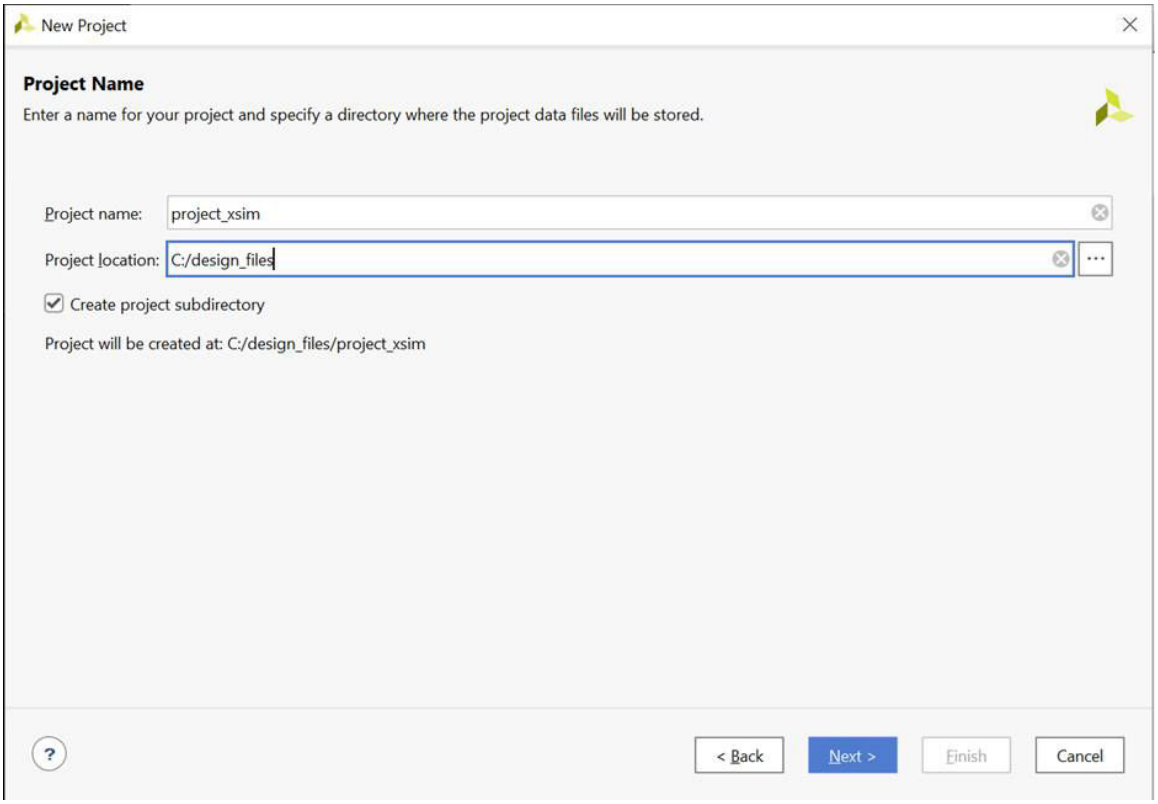
1. On Windows, launch the Vivado IDE by selecting **Start → All Programs → Xilinx Design Tools → Vivado 2021.x → Vivado 2021.x**

   (x denotes the latest version of Vivado 2021 IDE)

   Note: Your Vivado ML Editions installation might be called something other than Xilinx® Design Tools on the Start menu.

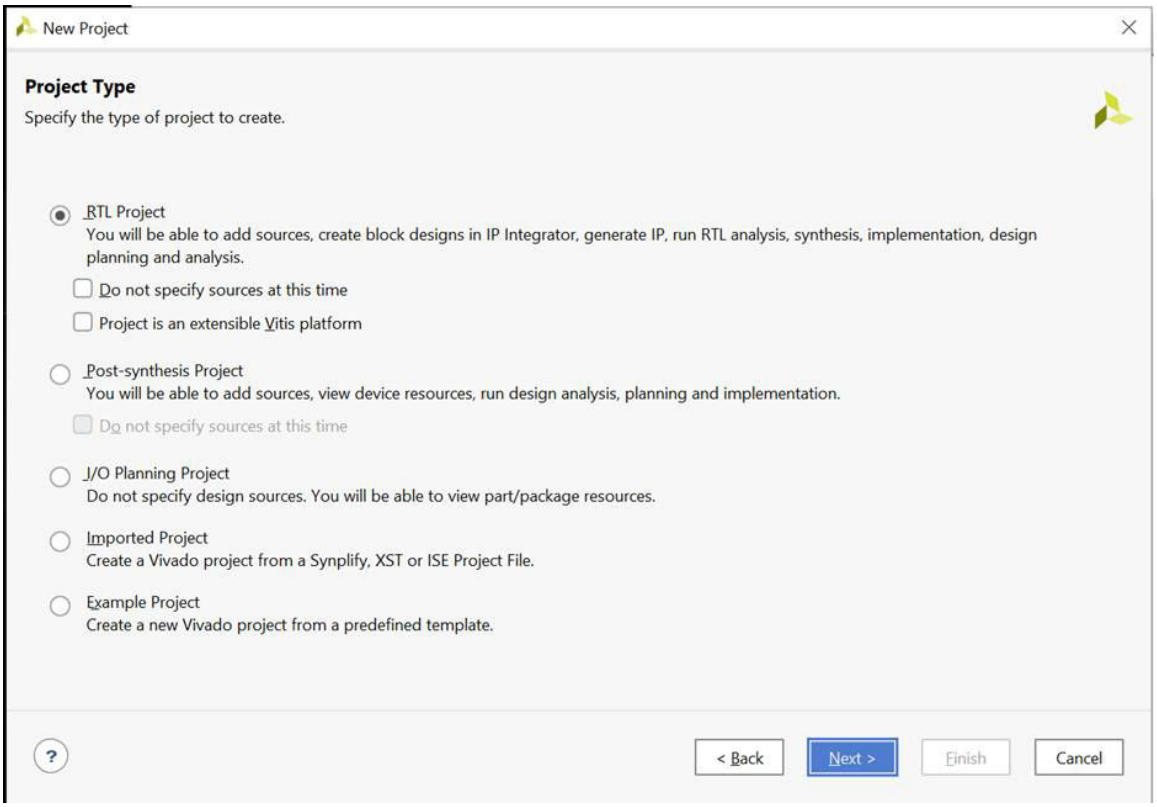
2. In the Vivado IDE Getting started page, click **Create Project**.
3. In the New project dialog box, click **Next** and enter a project name: `project_xsim`.

4. For the Project Location, browse to the folder containing the extracted tutorial data, `<Extract_Dir>`. Make sure to check the Create project subdirectory option and click **Next**.



Note: Create project subdirectory option is preselected.

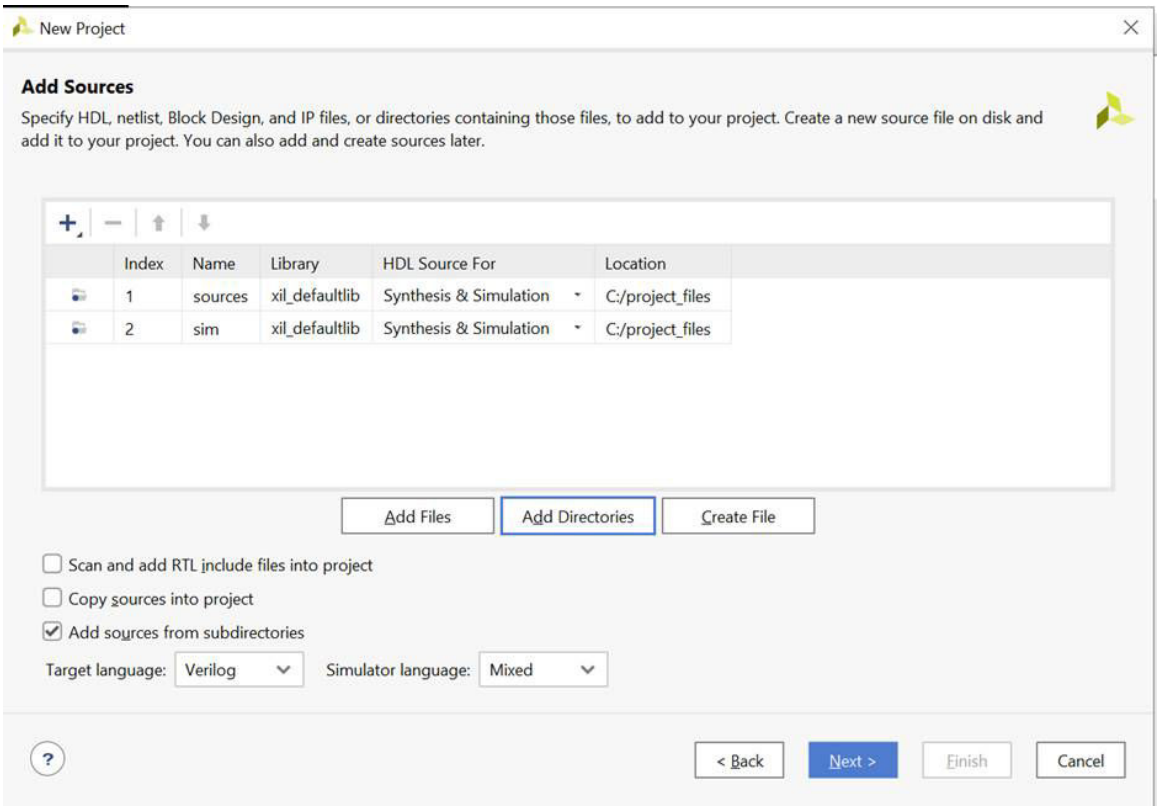
5. In the Project Type dialog box, select **RTL Project** and click **Next**.



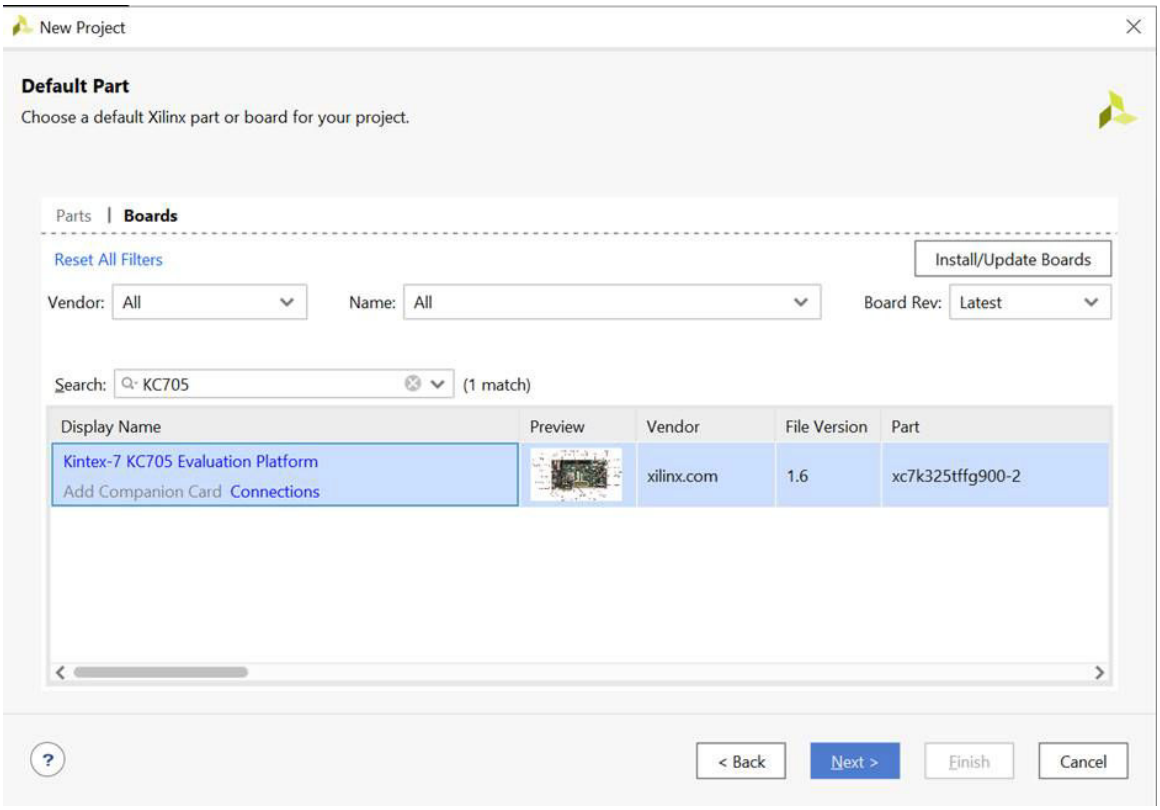
6. In the Add Source dialog box, click **Add Directories** and add the extracted tutorial design data:
   - `<Extract_Dir>/sources`
   - `<Extract_Dir>/sim`

   Note: You can press the Ctrl key to click and select multiple files or directories.
7. Set the Target Language to **Verilog** to indicate the netlist language for synthesis.
8. Set the Simulator Language to Mixed as shown in the following figure.

   The Simulator Language indicates which languages the logic simulator supports or requires. Vivado Design Site ensures the availability of simulation models of any IP cores in the design by using the available synthesis files to generate the required language-specific structural simulation model when generating output targets. For more information on working with IP cores and the Xilinx IP catalog, refer to the *Vivado Design Suite User Guide: Designing with IP* (UG896). You can also work through the *Vivado Design Suite Tutorial: Designing with IP* (UG939).
9. Click **Next**.
10. Click **Next** to bypass the Add Constraints dialog box.

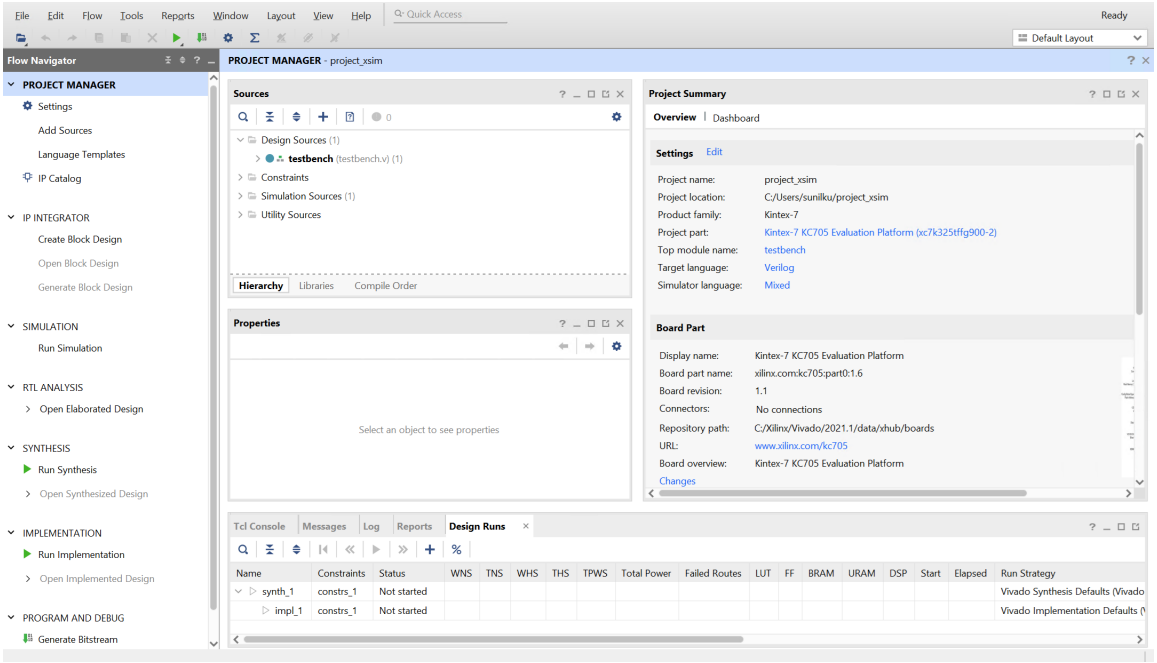


In the Default Part dialog box shown in the following figure, select **Boards**, and then select either **Kintex®-7 KC705 Evaluation Platform** for 7 series or **Kintex-UltraScale KCU105 Evaluation Platform** for UltraScale devices and click **Next**.



Note: Add sources from subdirectories option is preselected.

11. Review the New Project Summary dialog box.
12. Click **Finish** to create the project.

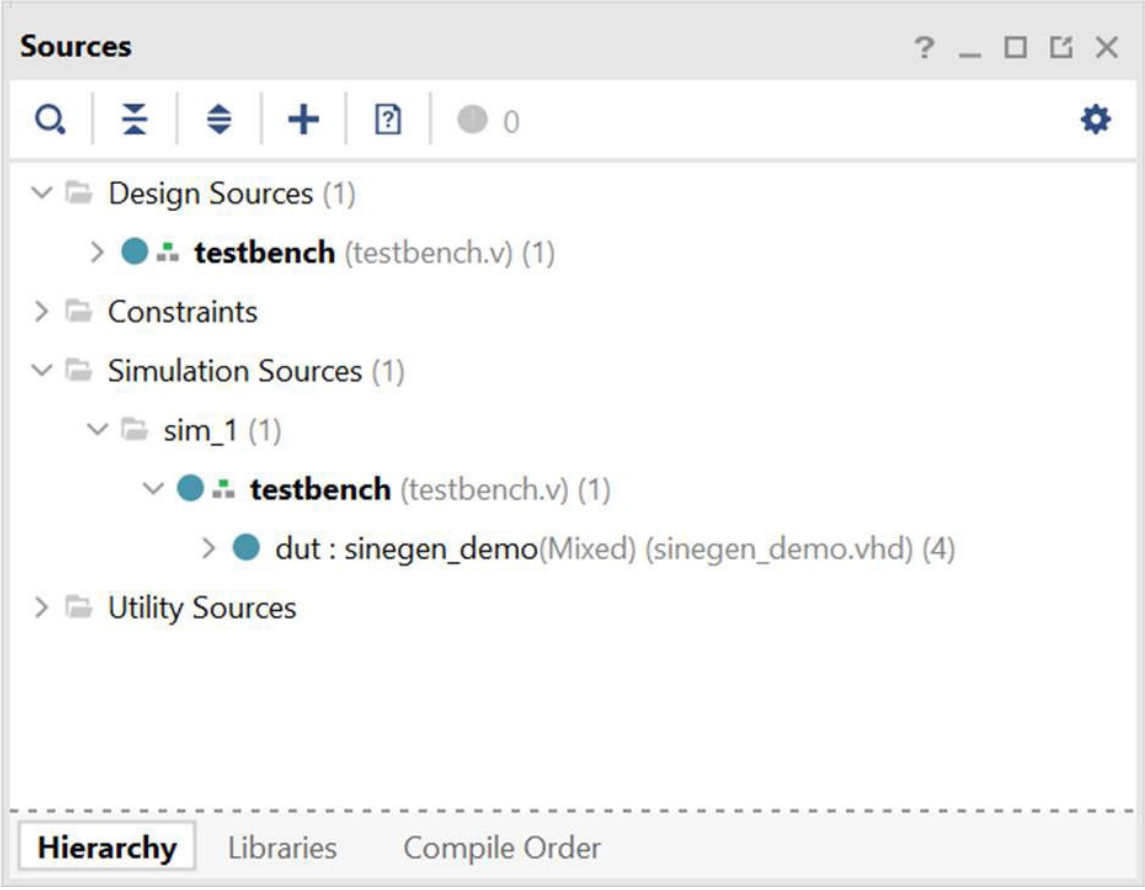


Vivado opens the new project in the Vivado IDE, using the default view layout

Step 2: Adding IP from the IP Catalog

The Sources window displays the source files that you have added during project creation. The Hierarchy tab displays the hierarchical view of the source files.

1. Click the icon in the Sources window to expand the folders as shown in the following figure. Expand all button can be used to view all the files at all levels of hierarchy.



Notice that the Sine wave generator (`sinegen.vhd`) references cells that are not found in the current design sources. In the Sources window, the missing design sources are marked by the missing source icon.

Note: The missing source icon is used to view only the missing sources. This is useful in viewing the missing sources in larger designs.

Now, add the `sine_high`, `sine_mid`, and `sine_low` modules to the project from the Xilinx IP catalog.

Adding Sine High

1. In the Flow Navigator, select the **IP Catalog** button.

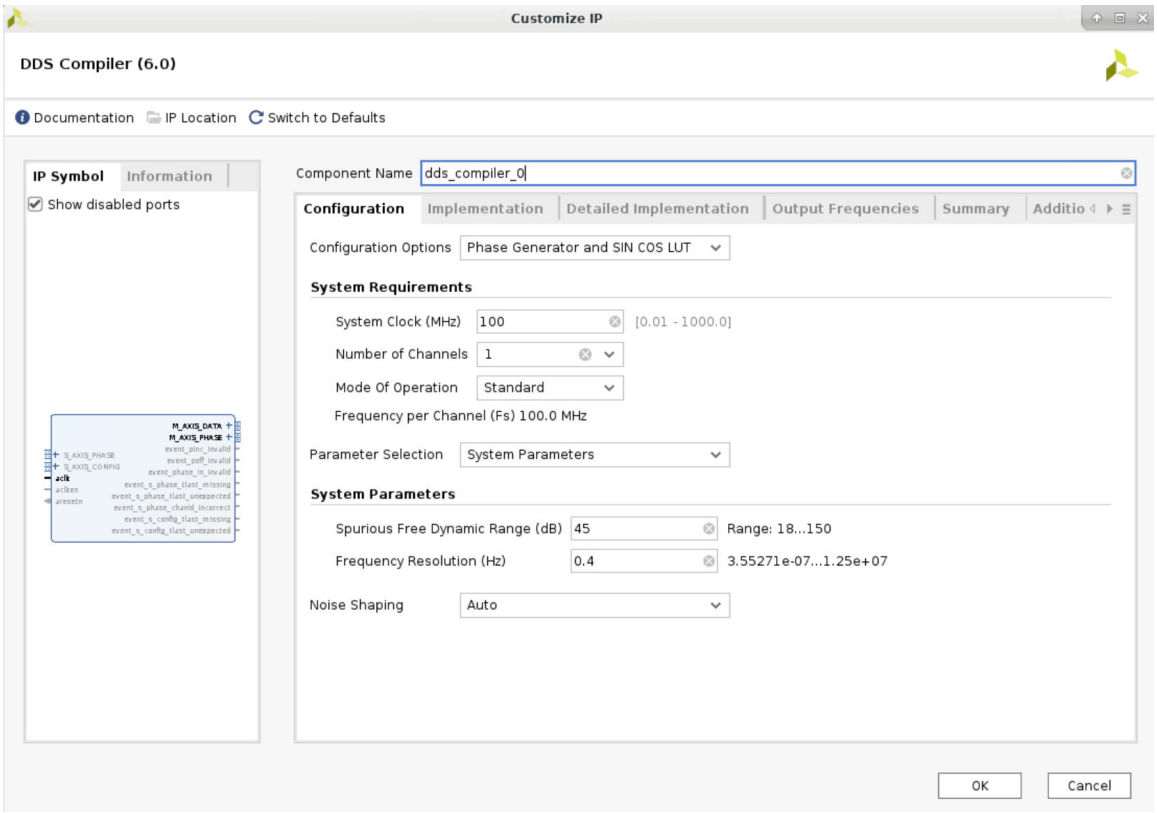
   The IP catalog opens in the graphical windows area. For more information on the specifics of the Vivado® IDE, refer to the*Vivado Design Suite User Guide: Using the Vivado IDE* (UG893).

2. In the search field of the IP catalog, type `DDS`.

   The Vivado IDE highlights the DDS Compilers in the IP catalog.

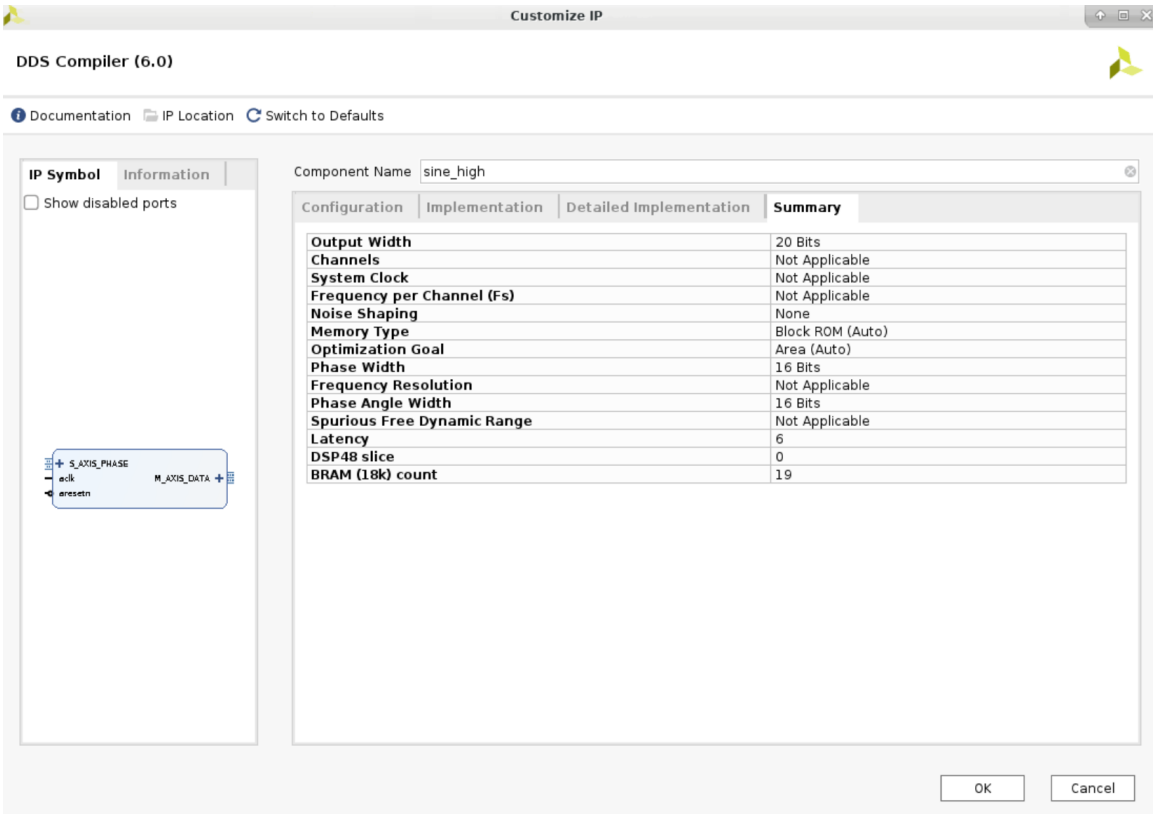
3. Under any category, double-click the **DDS Compiler**.

   The Customize IP wizard opens (following figure).

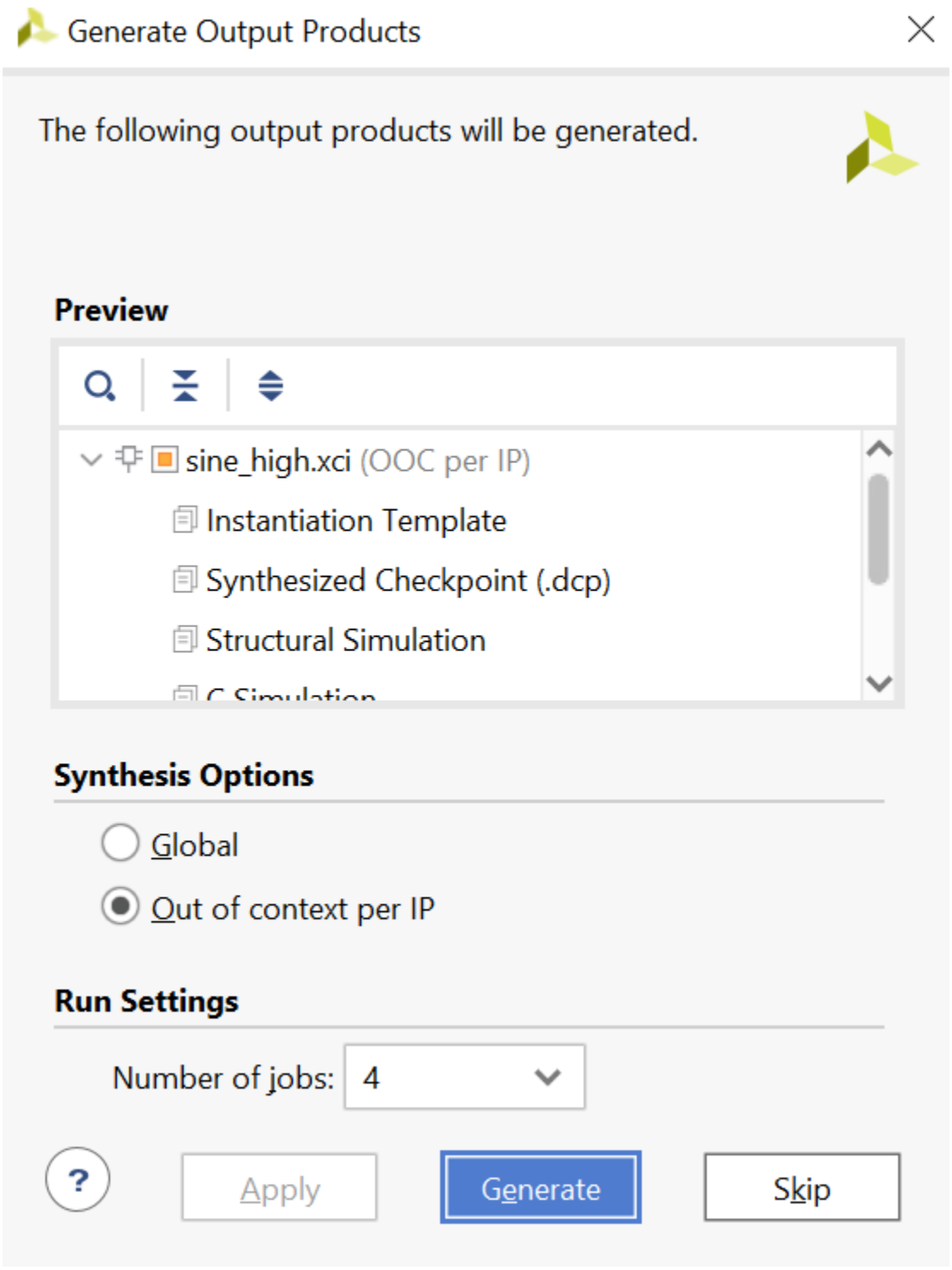


4. In the IP Symbol on the left, ensure that **Show disabled ports** is unchecked.
5. Specify the following on the Configuration tab:
   - Component Name: type `sine_high`
   - Configuration Options: select **SIN COS LUT** only
   - Noise Shaping: select None
   - Under Hardware Parameters, set Phase Width to 16 and Output Width to 20
6. On the Implementation tab, set Output Selection to **Sine**.
7. On the Detailed Implementation tab, set Control Signals to **ARESETn (active-Low)**.

8. On the Summary tab, review the settings and click **OK**.



When the `sine_high` IP core is added to the design, the output products required to support the IP in the design must be generated. The Generate Output Products dialog box displays, as shown in the following figure.

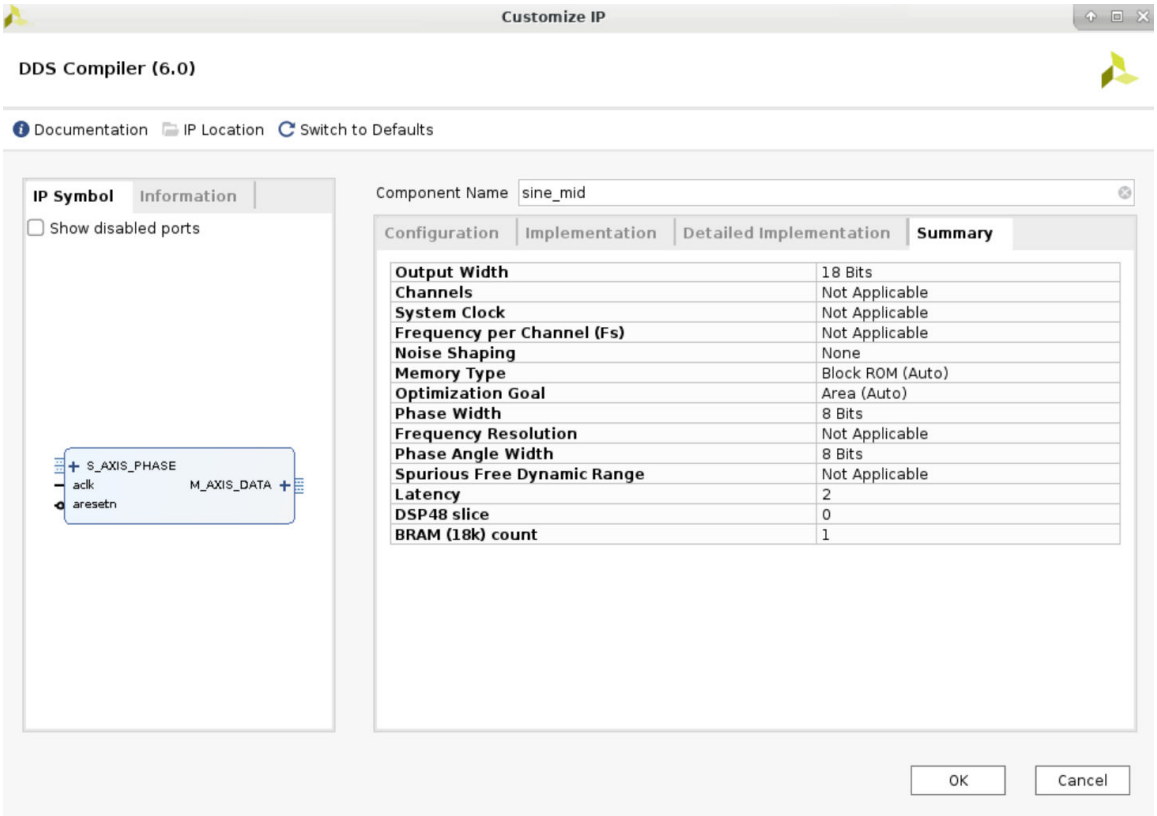


The output products allow the IP to be synthesized, simulated, and implemented as part of the design. For more information on working with IP cores and the Xilinx® IP catalog, refer to the *Vivado Design Suite User Guide: Designing with IP* (UG896). You can also work through the *Vivado Design Suite Tutorial: Designing with IP* (UG939).

9. Click **Generate** to generate the default output products for `sine_high`. A dialog box opens saying that the Out of context module run was launched for generating output products. Click **OK**.

Adding Sine Mid

1. In the IP catalog, double-click the **DDS Compiler IP** a second time.
2. Specify the following on the Configuration tab:
   - Component Name: type `sine_mid`
   - Configuration Options: select **SIN COS LUT** only
   - Noise Shaping: select **None**
   - Under Hardware Parameters, set the Phase Width to 8, and the Output Width to 18
3. On the Implementation tab, set the Output Selection to **Sine**.
4. On the Detailed Implementation tab, set Control Signals to **ARESETn (active-Low)**.
5. Select the **Summary** tab, review the settings and click **OK** as shown in the following figure:

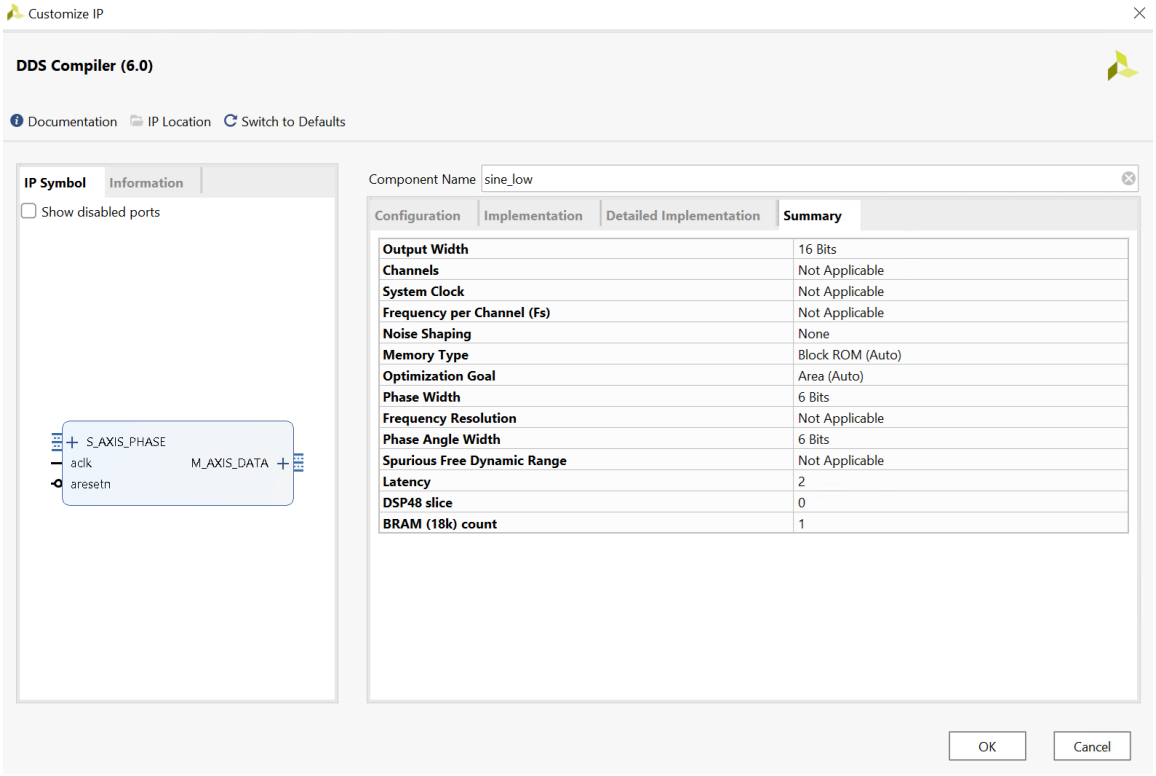


When the `sine_mid` IP core is added to the design, the Generate Output Products dialog box displays to generate the output products required to support the IP in the design.

6. Click **Generate** to generate the default output products for `sine_mid`. A dialog box opens saying that the Out of context module run was launched for generating output products. Click **OK**.

Adding Sine Low

1. In the IP catalog, double-click the **DDS Compiler IP** for the third time.
2. Specify the following on the Configuration tab:
   - Component Name: type `sine_low`
   - Configuration Options: select **SIN COS LUT** only
   - Noise Shaping: select **None**
   - Under Hardware Parameters, set the Phase Width to 6 and the Output Width to 16
3. On the Implementation tab, set the Output Selection to **Sine**.
4. On the Detailed Implementation tab, set Control Signals to **ARESETn (active-Low)**.
5. Select the **Summary** tab, review the settings as seen in the following figure, and click **OK**.



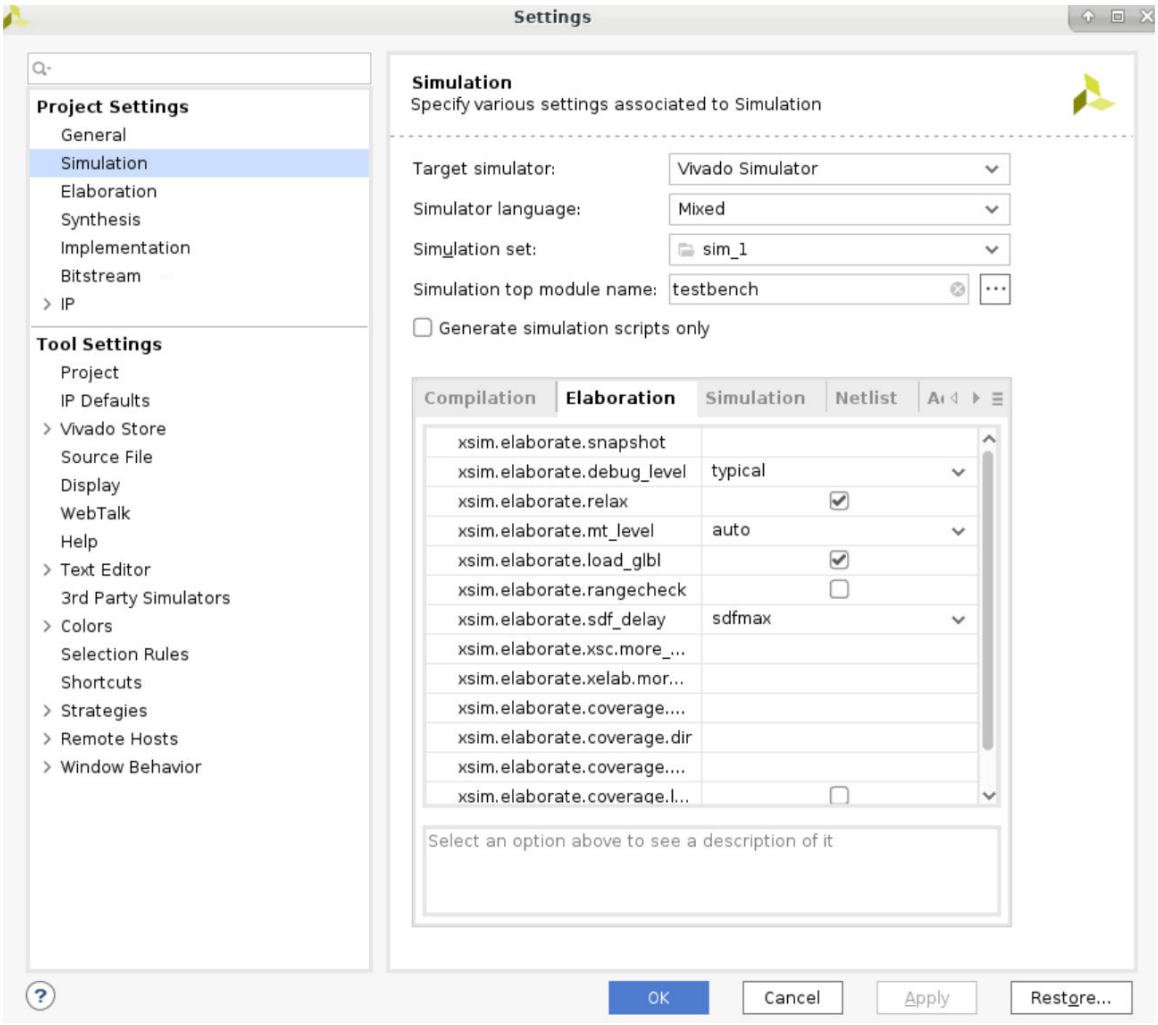
When the sine_low IP core is added to the design, the Generate Output Products dialog box displays to generate the output products required to support the IP in the design.

6. Click **Generate** to generate the default output products for `sine_low`. A dialog box opens saying that the Out of context module run was launched for generating output products. Click **OK**.

Step 3: Running Behavioral Simulation

After you have created a Vivado® project for the tutorial design, you set up and launch Vivado® simulator to run behavioral simulation. Set the behavioral simulation properties in Vivado® tools:

1. In the Flow Navigator, right-click **Simulation** and then click **Simulation Settings**. Alternatively, click **Settings** in the Flow Navigator under Project Manager to open the Settings window. Select **Simulation** from the Settings window. The following defaults are automatically set:
   - Simulation set: select sim_1
   - Simulation top-module name: set testbench
2. In the Elaboration tab (figure below), ensure that the debug level is set to typical, which is the default value.



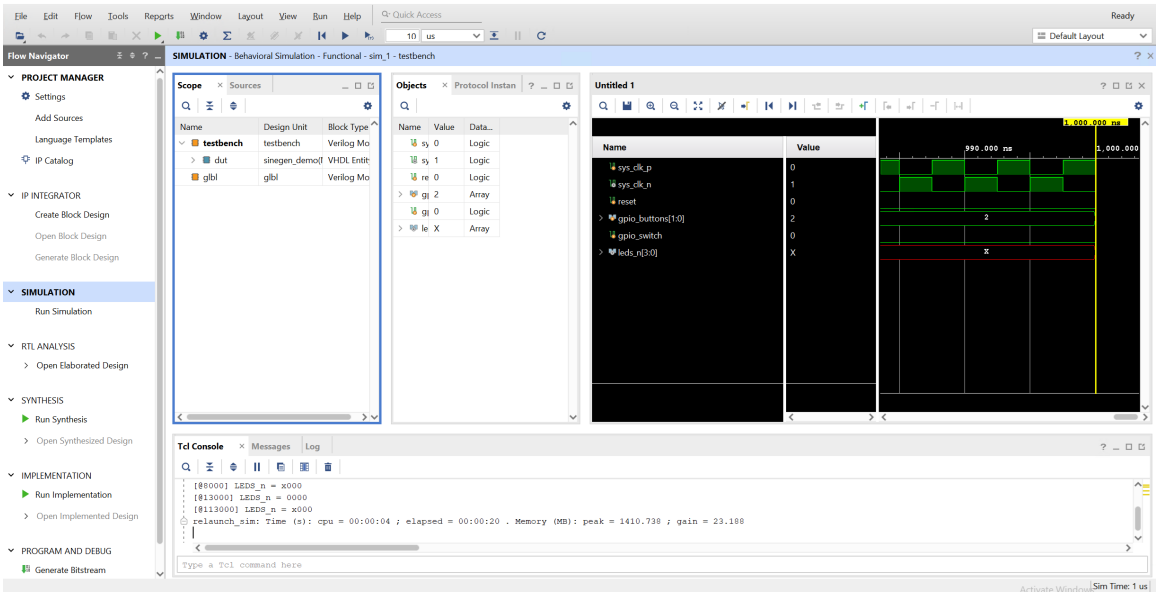
3. In the Simulation tab, observe that the Simulation Run Time is 1000ns.
4. Click **OK**.

   With the simulation settings properly configured, you can launch the Vivado simulator to perform a behavioral simulation of the design.
5. In the Flow Navigator, click **Run Simulation → Run Behavioral Simulation**.

   Functional and timing simulations are available post-synthesis and post-implementation. Those simulations are outside the scope of this tutorial.

   When you launch the Run Behavioral Simulation command, the Vivado tool runs `xvlog` and `xvhdl` to analyze the design and `xelab` in the background to elaborate and compile the design into a simulation snapshot, which the Vivado simulator can run. When that process is complete, the Vivado tool launches `xsim` to run the simulation.

   In the Vivado IDE, the simulator GUI opens after successfully parsing and compiling the design (figure below). By default, the top-level HDL objects display in the Waveform window.



Conclusion

In this lab, you have created a new Vivado® Design Suite project, added HDL design sources, added IP from the Xilinx® IP catalog and generated IP outputs needed for simulation, and then run behavioral simulation on the elaborated RTL design.

This concludes Lab 1. You can continue Lab 2 at this time by starting at Step 2: Displaying Signal Waveforms.

You can also close the simulation, project, and the Vivado IDE to start Lab 2 at a later time.

1. Click **File → Close Simulation** to close the open simulation.
2. Select **OK** if prompted to confirm closing the simulation.
3. Click **File → Close Project** to close the open project.
4. Click **File → Exit** to exit the Vivado tool.

Lab 2: Debugging the Design

The Vivado® simulator GUI contains the Waveform window, and Object and Scope Windows. It provides a set of debugging capabilities to quickly examine, debug, and fix design problems. See the *Vivado Design Suite User Guide: Logic Simulation* (UG900) for more information about the GUI components.

In this lab, you:

- Enable debug capabilities
- Examine a design bug
- Use debug features to find the root cause of the bug
- Make changes to the code
- Re-compile and re-launch the simulation

Step 1: Opening the Project

This lab continues from the end of Lab #1 in this tutorial. You must complete Lab #1 prior to beginning Lab #2. If you closed the Vivado IDE, or the tutorial project, or the simulation at the end of Lab #1, you must reopen them.

Start by loading the Vivado Integrated Design Environment (IDE) by selecting **Start → All Programs → Xilinx Design Tools → Vivado 2021.x → Vivado 2021.x**.

***Note*:**

1. Your Vivado ML Editions installation might be called something other than Xilinx Design Tools on the Start menu.
2. As an alternative, click the **Vivado 2021.x** Desktop icon to start the Vivado IDE.

The Vivado IDE opens. Now, open the project from Lab #1, and run behavioral simulation.

1. From the main menu, click **File → Project → Open Recent** and select `project_xsim`, which you saved in Lab #1.
2. After the project has opened, from the Flow Navigator click **Run Simulation → Run Behavioral Simulation**.

The Vivado simulator compiles your design and loads the simulation snapshot.

Step 2: Displaying Signal Waveforms

In this section, you examine features of the Vivado simulator GUI that help you monitor signals and analyze simulation results, including:

- Running and restarting the simulation to review the design functionality, using signals in the Waveform window and messages from the testbench shown in the Tcl console.
- Adding signals from the testbench and other design units to the Waveform window so you can monitor their status.
- Adding groups and dividers to better identify signals in the Waveform window.
- Changing signal and wave properties to better interpret and review the signals in the Waveform window.
- Using markers and cursors to highlight key events in the simulation and to perform zoom and time measurement features.
- Using multiple waveform configurations.

Add and Monitor Signals

The focus of the tutorial design is to generate sine waves with different frequencies. To observe the function of the circuit, you monitor a few signals from the design. Before running simulation for a specified time, you can add signals to the wave window to observe the signals as they transition to different states over the course of the simulation.

By default, the Vivado® simulator adds simulation objects from the testbench to the Waveform window. In the case of this tutorial, the following testbench signals load automatically:

- Differential clock signals (`sys_clk_p` and `sys_clk_n`). This is a 200 MHz clock generated by the testbench and is the input clock for the complete design.
- Reset signal (`reset`). Provides control to reset the circuit.
- GPIO buttons (`gpio_buttons[1:0]`). Provides control signals to select different frequency sine waves.
- GPIO switch (`gpio_switch`). Provides a control switch to enable or disable debouncer logic.
- LEDs (`leds_n[3:0]`). A placeholder bus to display the results of the simulation.

You add some new signals to this list to monitor those signals as well.

If necessary, in the Scopes window, click the ⌄ sign to expand the testbench. (It might be expanded by default.)

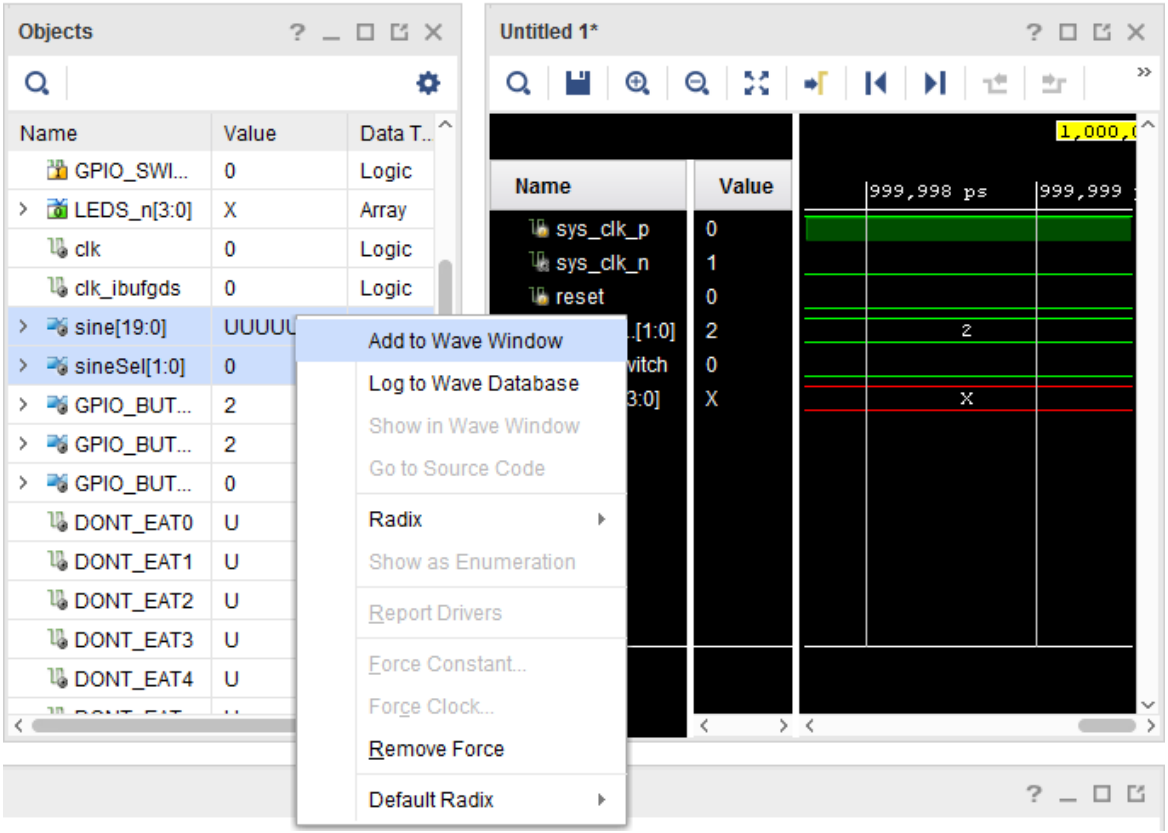
An HDL scope, or scope, is defined by a declarative region in the HDL code, such as a module, function, task, process, or named blocks in Verilog. VHDL scopes include entity/architecture definitions, blocks, functions, procedures, and processes.

1. In the Scopes window, click to select the **dut** object.

   The current scope of the simulation changes from the whole testbench to the selected object. The Objects window updates with all the signals and constants of the selected scope, as shown in figure below.

2. From the Objects window, select signals `sine[19:0]` and `sineSel[1:0]` and add them into Wave Configuration window using one of the following methods:
   - Drag and drop the selected signals into the Waveform window.
   - Right-click on the signal to open the popup menu, and select **Add to Wave Window**.

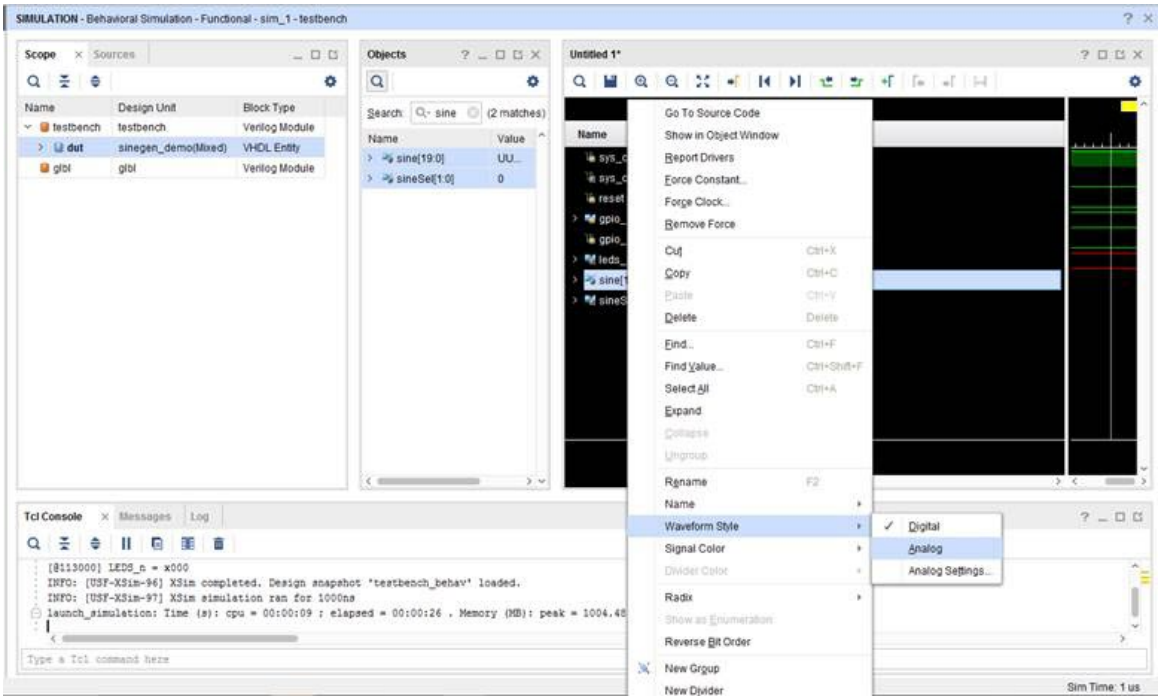
***Note*:** You can select multiple signals by holding down the CTRL key during selection.



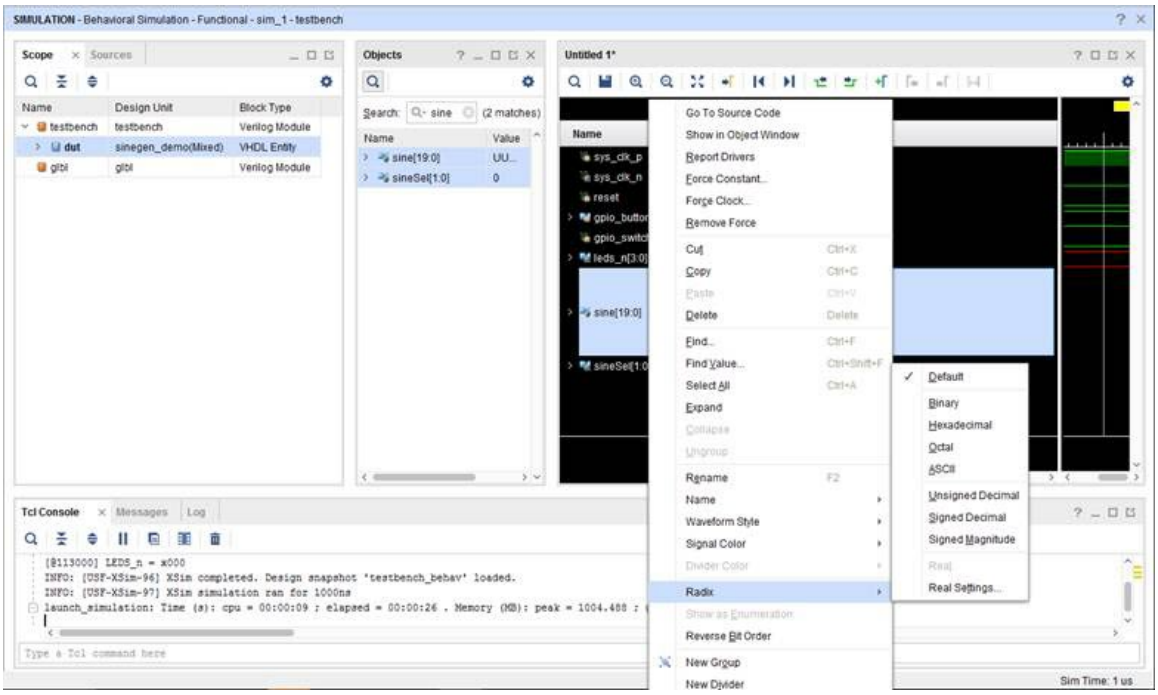
Step 3: Using the Analog Wave Viewer

The sine[19:0] signals you are monitoring are analog signals, which you can view better in Analog wave mode. You can choose to display a given signal as Digital or Analog in the Waveform window.

1. In the Waveform window, select the `sine[19:0]` signal.
2. Right-click to open the popup menu, and select **Waveform Style → Analog**, as shown in the figure below.



3. Right-click to open the popup menu again, and select **Radix → Signed Decimal** as shown in the figure below.



Logging Waveforms for Debugging

The Waveform window lets you review the state of multiple signals as the simulation runs. However, due to its limited size, the number of signals you can effectively monitor in the Waveform window is limited. To identify design failures during debugging, you might need to trace more signals and objects than can be practically displayed in the Waveform window. You can log the waveforms for signals that are not displayed in the Waveform window, by writing them to the simulation waveform database (WDB). After simulation, you can review the transitions on all signals captured in the waveform database file.

In the Scope window, right-click on **dut** under testbench. Click `Log to Wave Database` from the options list. Select **Objects in Scope** option. The specified signals are written to a waveform database.

Using the Tcl Command

Enable logging of the waveform for the specified HDL objects by entering the following command in the Tcl console:

```
log_wave [get_objects /testbench/dut/*] [get_objects /testbench/dut/
U_SINEGEN/*]
```

Note: See the *Vivado Design Suite Tcl Command Reference Guide* (UG835) for more information on the `log_wave` command.

This command enables signal dumping for the specified HDL objects, `/testbench/dut/*` and `/testbench/dut/U_SINEGEN/*`.

Note: * symbol specifies all the HDL objects in a scope.

The `log_wave` command writes the specified signals to a waveform database, which is written to the simulation folder of the current project:

```
<project_name>/<project_name>.sim/sim_1/behav/xsim
```

Step 4: Working with the Waveform Window

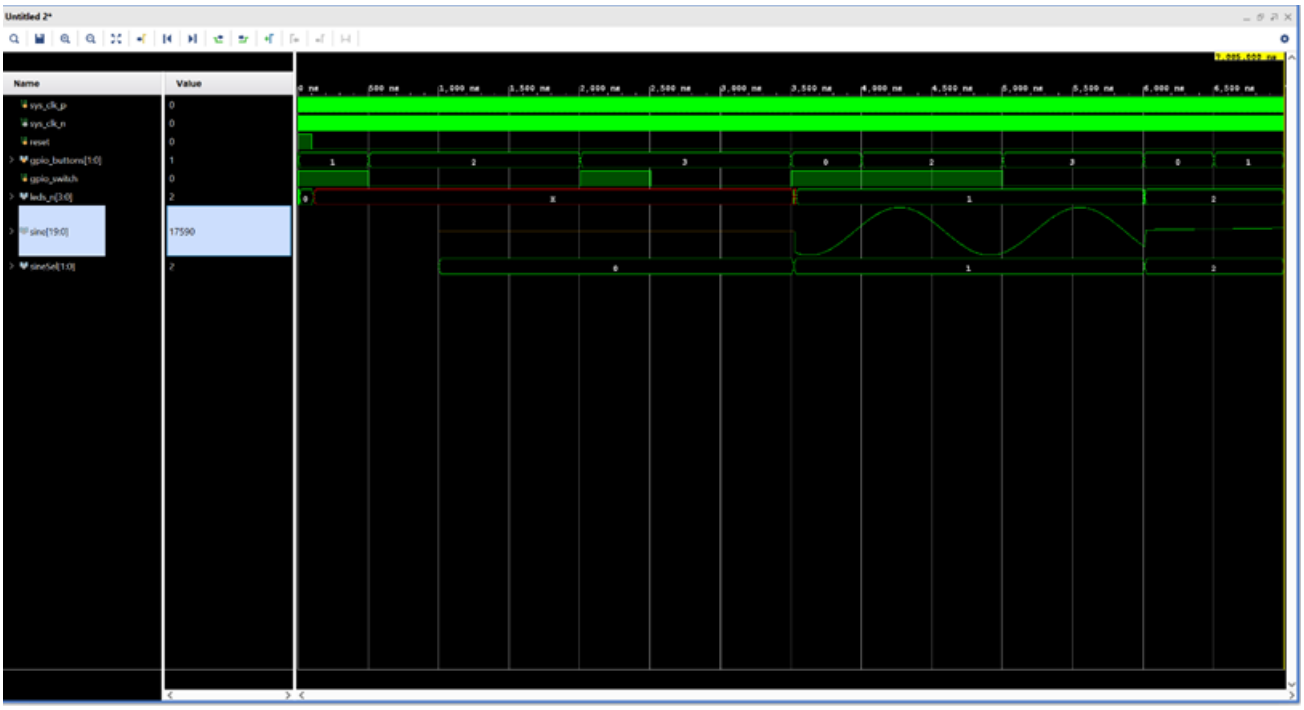
Now that you have configured the simulator to display and log signals of interest into the waveform database, you are ready to run the simulator again.

1. Run the simulation by clicking the **Run All** button.

   Observe the sine signal output in the waveform. The Wave window can be unlocked from Main window layout to view it as standalone.

2. Click the **Float** button in the title bar of the Waveform Configuration window.
3. Click the **Zoom Fit** button to display the whole time spectrum in the Waveform Configuration window.

Notice that the low frequency sine output is incorrect. You can view the waveform in detail by zooming into the Waveform window. When you zoom into the waveform, you can use the horizontal and vertical scroll bars to pan down the full waveform.



As seen in the figure above, when the value of sineSel is 0, which indicates a low frequency sine selection, the analog `sine[19:0]` output is not a proper sine wave, indicating a problem in the design or the testbench.

Grouping Signals

Next, you add signals from other design units to better analyze the functionality of the whole design. When you add signals to the Waveform window, the limited size of the window makes it difficult to display all signals at the same time. Reviewing all signals would require the use of the vertical scroll bar, making the review process difficult.

You can group related signals together to make viewing them easier. With a group, you can display or hide associated signals to make the Waveform window less cluttered, and easier to understand.

1. In the Waveform window, select all signals in the testbench unit: `sys_clk_p`, `sys_clk_n`, `reset`, `gpio_buttons`, `gpio_switch`, and `leds_n`.

   Note: Press and hold the Ctrl key, or Shift key, to select multiple signals.

2. With the signals selected right-click to open the popup menu and select **New Group**. Rename it as TB Signals.

   The Vivado® simulator creates a collapsed group in the waveform configuration window. To expand the group, click ❯ to the left of the group name.

3. Create another signal group called DUT Signals to group signals `sine[19:0]` and `sine_sel[1:0]`.

You can add or remove signals from a group as needed. Cut and paste signals from the list of signals in the Waveform window, or drag and drop a signal from one group into another.

You can also drag and drop a signal from the Objects window into the Waveform window, or into a group.

You can ungroup all signals, thereby eliminating the group. Select a group, right-click to open the popup menu and select **Ungroup**.

To better visualize which signals belong to which design units, add dividers to separate the signals by design unit.

Adding Dividers

Dividers let you create visual breaks between signals or groups of signals to more easily identify related objects.

1. In the Waveform window, right-click to open the popup menu and select **New Divider**. The Name dialog box opens to let you name the divider you are adding to the Waveform window.
2. Add two dividers named:
   - `Testbench`
   - `SineGen`
3. Move the `SineGen` divider above the `DUT Signals` group.

TIP: *You can change divider names at any time by highlighting the divider name and selecting the Rename command from the popup menu, or change the color with Divider Color.*

Adding Signals from Sub-modules

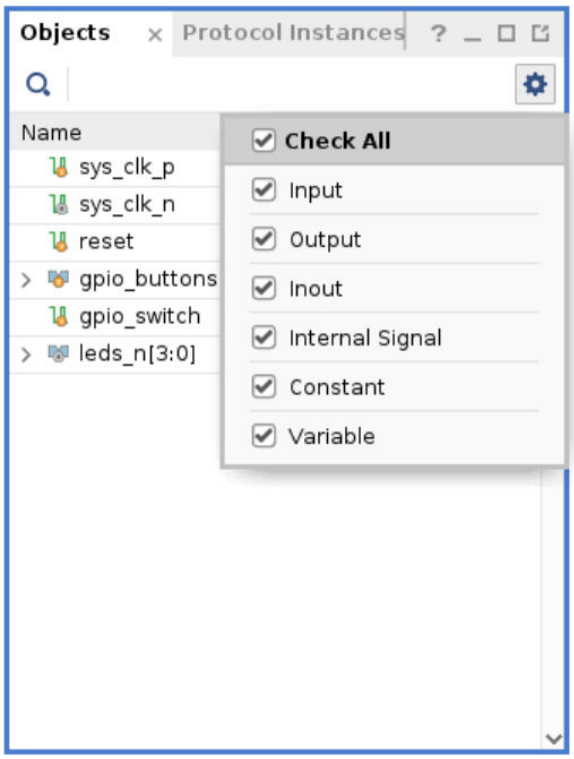
You can also add signals from different levels of the design hierarchy to study the interactions between these modules and the testbench. The easiest way to add signals from a sub-module is to filter objects and then select the signals to add to the Waveform view.

Add signals from the instantiated `sine_gen_demo module` (DUT) and the `sinegen` module (U_SINEGEN).

1. In the Scopes window, select and expand the `Testbench`, then select and expand `DUT`.

   Simulation objects associated with the currently selected scope display in the Objects window.

   By default, all types of simulation objects display in the Objects window. However, you can limit the types of objects displayed by selecting the object filters at the top of the Objects window. The following figure shows the Objects window with the Input and Output port objects enabled, and the other object types are disabled. Move the cursor to hover over a button to see the tooltip for the object type.



2. Use the Objects window toolbar to enable and disable the different object types.

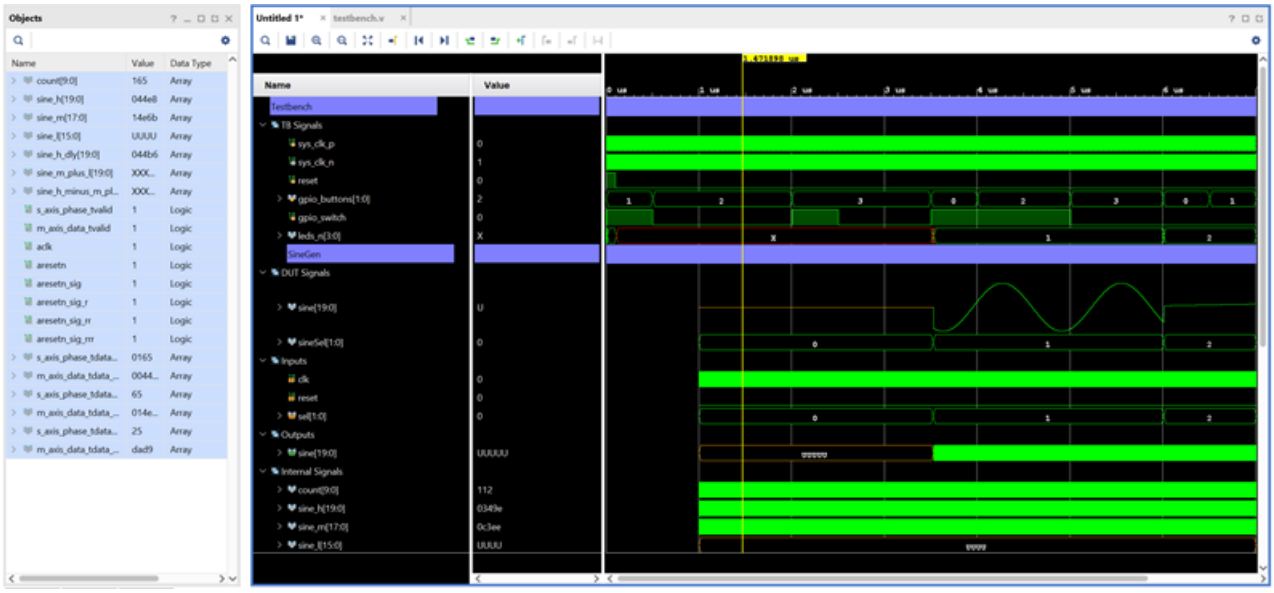
   The types of objects that can be filtered in the Objects window include Input, Output, Inout ports, Internal Signals, Constants, and Variables.

3. In the Scopes window, select the `U_SINEGEN` design unit.
4. In the Waveform window, right-click in the empty space below the signal names, and use the New Group command to create three new groups called `Inputs`, `Outputs`, and `Internal Signals`.

   TIP: *If you create the group on top of, or containing, any of the current objects in the Waveform window, simply drag and drop the objects to separate them as needed.*

5. In the Objects window, select the Input filter to display the Input objects.
6. Select the Input objects in the Objects window, and drag and drop them onto the Input group you created in the Waveform window.

Repeat steps 5 and 6 above to filter the Output objects and drag them onto the Output group, and filter the Internal Signals and drag them onto the Internal Signals group, as shown in the following figure.



Step 5: Changing Signal Properties

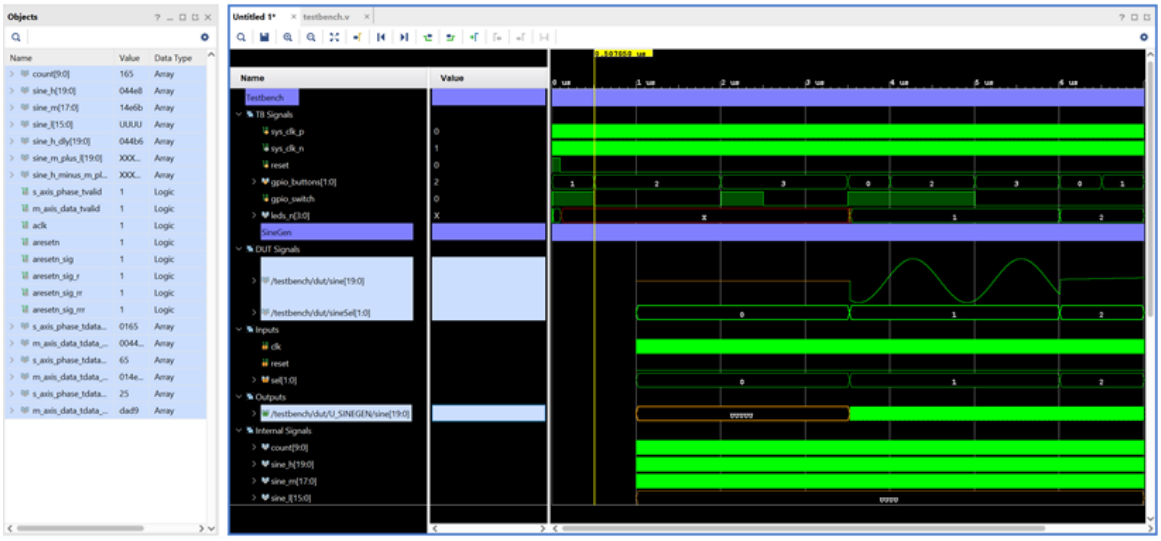
You can also change the properties of some of the signals shown in the Waveform window to better visualize the simulation results.

Viewing Hierarchical Signal Names

By default, the Vivado® simulator adds signals to the waveform configuration using a short name with the hierarchy reference removed. For some signals, it is important to know to which module they belong.

1. In the Waveform window, hold Ctrl and click to select the `sine[19:0]` and `sineSel[1:0]` signals listed in the `DUT` signals group, under the `SineGen` divider.
2. Hold Ctrl, and click to select the `sine[19:0]` signals listed in the `Outputs` group, under the `SineGen` divider.
3. Right-click in the **Waveform** window to open the popup menu, and select the **Name → Long** command.

   The displayed name changes to include the hierarchical path of the signal. You can now see that the `sine[19:0]` signals under the DUT Signals group refers to different objects in the design hierarchy than the `sine[19:0]` signals listed under the `Outputs` group. See the figure below.



Viewing Signal Values

You can better understand some signal values if they display in a different radix format than the default, for instance, binary values instead of hexadecimal values. The default radix is Hexadecimal unless you override the radix for a specific object.

Supported radix values are Binary, Hexadecimal, Octal, ASCII, Signed and Unsigned decimal. You can set any of the above values as Default using Default Radix option.

1. In the Waveform window, select the following signals:

   `s_axis_phase_tdata_sine_high`, `s_axis_phase_tdata_sine_mid` and `s_axis_phase_tdata_sine_low`.

2. Right-click to open the popup menu, and select **Radix → Binary**.

   The values on these signals now display using the specified radix

Step 6: Saving the Waveform Configuration

You can customize the look and feel of the Waveform window, and then save the Waveform configuration to reuse in future simulation runs. The Waveform configuration file defines the displayed signals, and the display characteristics of those signals.

1. In the Waveform window, click the **Settings** button on the title bar menu.

   The Waveform Options dialog box opens to the General tab.

2. Ensure the Default Radix is set to Hexadecimal.

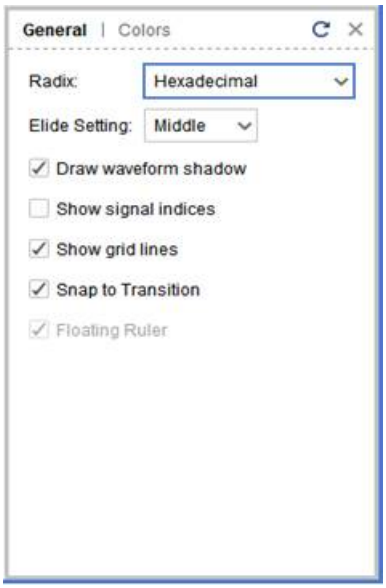
This defines the default number format for all signals in the Waveform window. The radix can also be set for individual objects in the Waveform window to override the default.

3. Select the **Draw Waveform Shadow**, as shown in the following figure, to enable or disable the shading under the signal waveform.

   By default, a waveform is shaded under the high transitions to make it easier to recognize the transitions and states in the Waveform window.

   You can also enable or disable signal indices, so that each signal or group of signals is identified with an index number in the Waveform window.

4. Check or uncheck the **Show signal indices** check box to enable or disable the signal list numbering.



5. Check or uncheck the **Show grid lines** check box to enable or disable the grid lines in waveform window.
6. Check the **Snap to Transition** check box to snap the cursor to transition edges.
7. In the Waveform Options dialog box, select the Colors view.

   Examine the Waveform Color Options dialog box. You can configure the coloring for elements of the Waveform window to customize the look and feel. You can specify custom colors to display waveforms of certain values, so you can quickly identify signals in an unknown state, or an uninitialized state.

   The Waveform window configures with your preferences. You can save the current waveform configuration so it is available for use in future Vivado® simulation sessions.

   By default, the Vivado simulator saves the current waveform configuration setting as `testbench_behav.wcfg`.

8. In the Waveform window sidebar menu, select the **Save Wave Configuration** button .
9. Save the Wave Configuration into the project folder with the filename `tutorial_1.wcfg`.

10. Click **Yes**. The file is added to the project simulation fileset, `sim_1`, for archive purposes.

> **TIP:** *You can also load a previously saved waveform configuration file using the **File → Simulation Waveform → Open Configuration** command.*

Working with Multiple Waveform Configurations

You can also have multiple Waveform windows, and waveform configuration files open at one time. This is useful when the number of signals you want to display exceeds the ability to display them in a single window. Depending on the resolution of the screen, a single Waveform window might not display all the signals of interest at the same time. You can open multiple Waveform windows, each with their own set of signals and signal properties, and copy and paste between them.

1. To add a new Waveform window, select **File → Simulation Waveform → New Configuration**.

   An untitled Waveform window opens with a default name. You can add signals, define groups, add dividers, set properties and colors that are unique to this Waveform window.

2. Select signal groups in the first Waveform window by pressing and holding the Ctrl key, and selecting the following groups: `Inputs`, `Outputs`, and `Internal Signals`.
3. Right-click to open the popup menu, and select **Copy**, or use the shortcut Ctrl+C on the selected groups to copy them from the current Waveform window.
4. Select the new Waveform window to make it active.
5. Right-click in the Waveform window and select **Paste**, or use the shortcut Ctrl+V to paste the signal groups into the prior Waveform window.
6. Select **File → Simulation Waveform → Save Configuration** or click the **Save Wave Configuration** button, and save the waveform configuration to a file called `tutorial_2.wcfg`.
7. When prompted to add the waveform configuration to the project, select **No**.
8. Click the icon to close the new Waveform window.

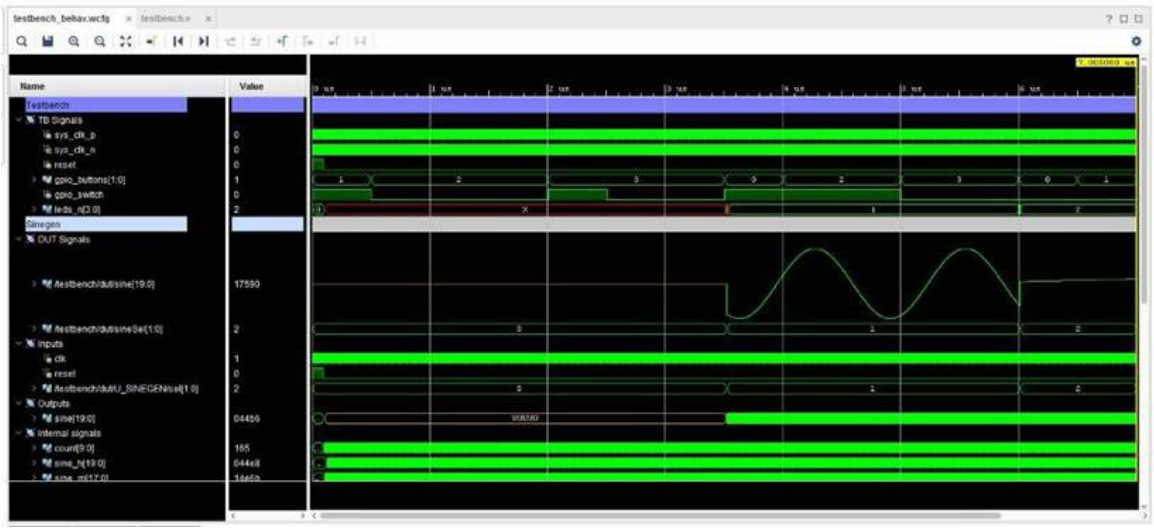
Step 7: Re-Simulating the Design

With the various signals, signal groups, dividers, and attributes you have added to the Waveform window, you are now ready to simulate the design again.

1. Click the **Restart** button  to reset the circuit to its initial state.
2. Click the **Run All** button  .

The simulation runs for about 7005 ns. If you do not restart the simulator prior to executing the Run All command, the simulator runs continuously until interrupted.

3. After the simulation is complete, click the **Zoom Fit** button to see the whole simulation timeline in the Waveform window. Figure below shows the current simulation results.



Step 8: Using Cursors, Markers, and Measuring Time

The Finite State Machine (U_FSM) module used in the top-level of the design generates three different sine-wave select signals for specific outputs of the `SineGen` block. You can identify these different wave selections better using Markers to highlight them.

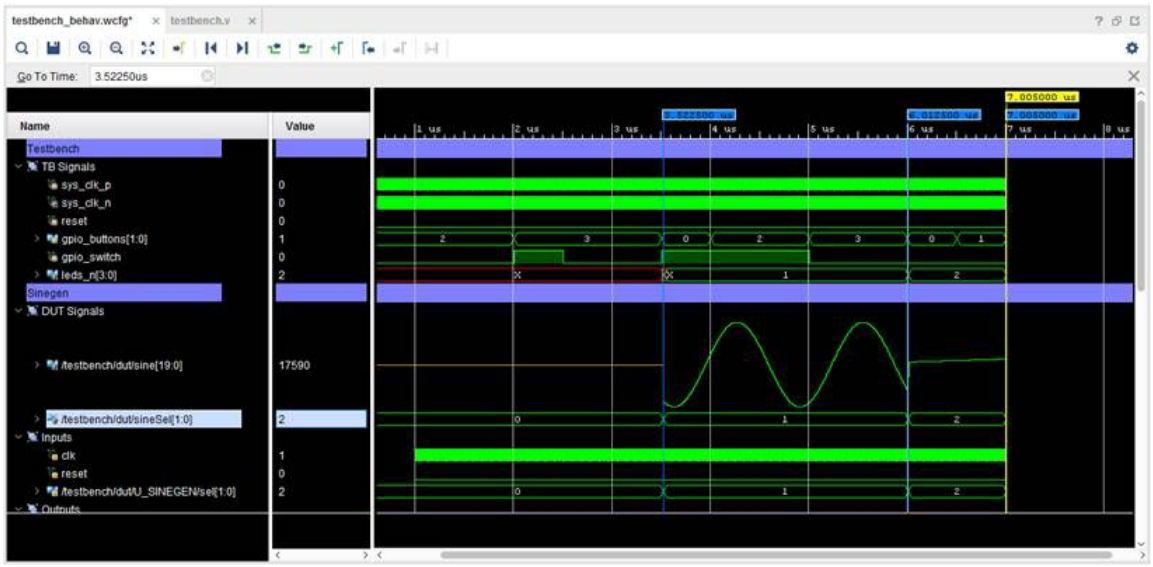
1. In the Waveform window select the `/testbench/dut/sineSel[1:0]` signal, as shown in the following figure.
2. In the waveform sidebar menu, click the **Go to Time 0** button .

   The current marker moves to the start of the simulation run.
3. Enable the **Snap to Transition** check box in the General tab of settings window to snap the cursor to transition edges.
4. From the waveform toolbar menu, click the **Next Transition** button.

   The current marker moves to the first value change of the selected `sineSel[1:0]` signal, at 3.5225 microseconds.
5. Click the **Add Marker** button.
6. Search for all transitions on the `sineSel` signal, and add markers at each one.

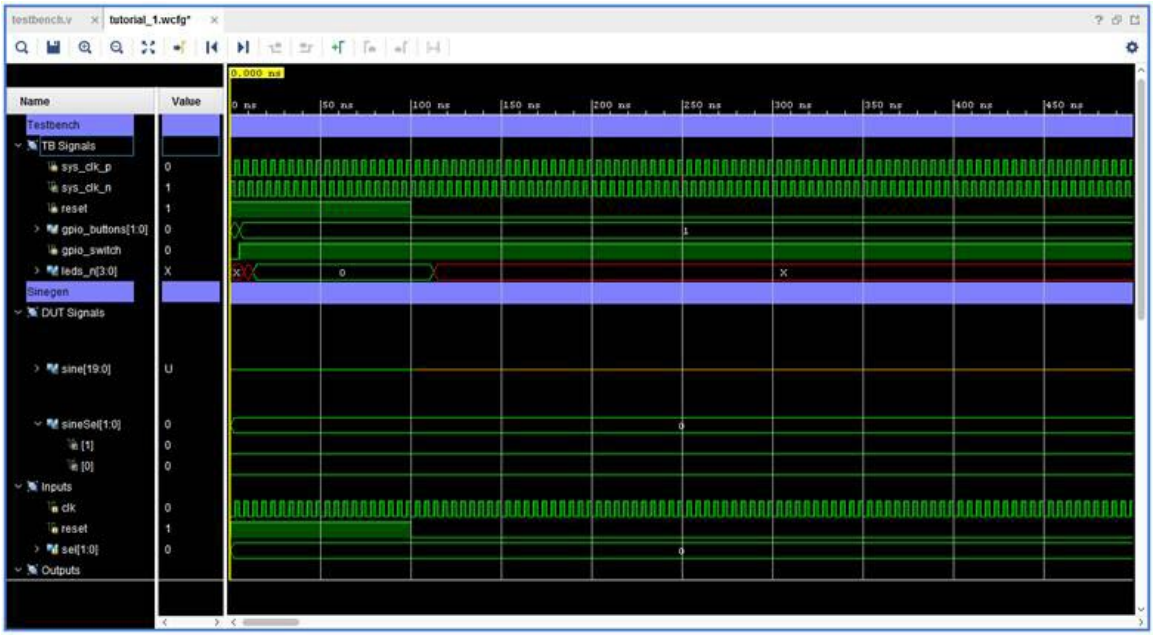
With markers identifying the transitions on `sineSel`, the Waveform window should look similar to the following figure. As previously observed, the low frequency signals are incorrect when the `sinSel` signal value is 0.

You can also use the main Waveform window cursor to navigate to different simulation times, or locate value changes. In the next steps, you use this cursor to zoom into the Waveform window when the `sineSel` is 0 to review the status of the output signal, `sine[19:0]`, and identify where the incorrect behavior initiates. You also use the cursor to measure the period of low frequency wave control.



TIP: *By default, the Waveform window displays the time unit in microseconds. However, you can use whichever measurement you prefer while running or changing current simulation time, and the Waveform window adjusts accordingly.*

7. In the Waveform window, click the **Go to Time 0** button , then click the Zoom in button repeatedly to zoom into the beginning of the simulation run.
8. Continue to zoom in the Waveform window as needed, until you can see the reset signal asserted low, and you can see the waveform of the clock signals, `sys_clk_p` and `sys_clk_n`, as seen in the following figure.



The Waveform window zooms in or out around the area centered on the cursor.

9. Place the main Waveform window cursor on the area by clicking at a specific time or point in the waveform.

   You can also click on the main cursor, and drag it to the desired time.

10. Because 0 is the initial or default FSM output, move the cursor to the first posedge of `sys_clk_p` after reset is asserted low, at time 102.5 ns, as seen in the following figure.

    You can use the Waveform window to measure time between two points on the timeline.

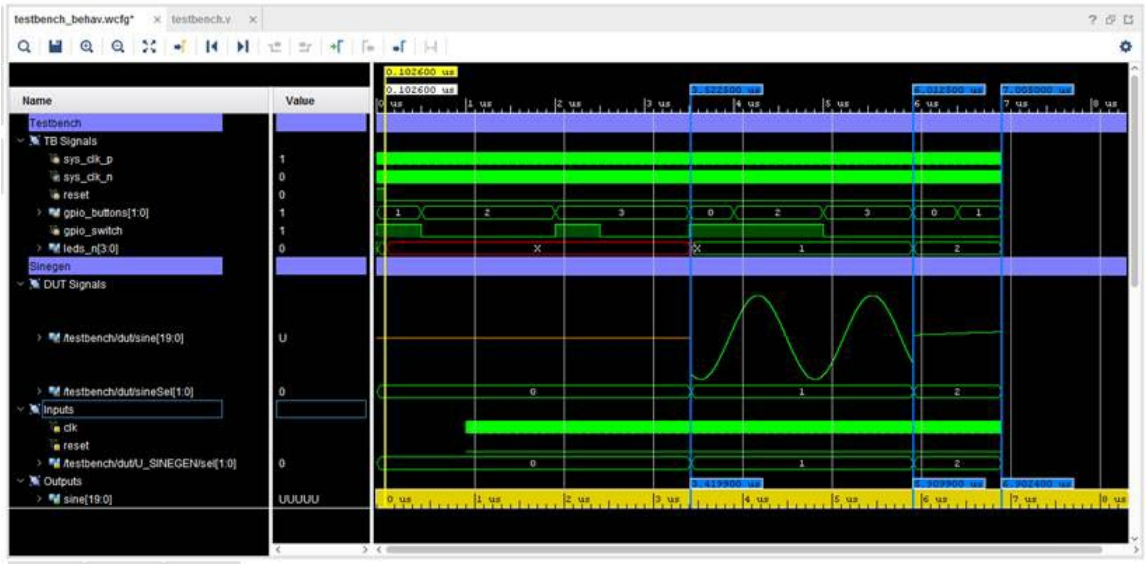
11. Place a marker at the time of interest, 102.5 ns, by clicking the **Add Marker** button .

12. Click to select the marker.

    The Floating Ruler option that is available in the General tab of waveform Settings displays a ruler at the bottom of the Waveform window useful for measuring time between two points. Use the floating ruler to measure the sineSel control signal period, and the corresponding `output_sine[19:0]` values during this time frame.

    When you select the marker, a floating ruler opens at the bottom of the Waveform window, with time 0 on the ruler positioned at the selected marker. As you move the cursor along the timeline, the ruler measures the time difference between the cursor and the marker.

> **TIP:** *Enable the Floating Ruler checkbox from the General tab of Waveform Settings, if the ruler does not appear when you select the marker.*



You can move the cursor along the timeline in a number of ways. You can scroll the horizontal scroll bar at the bottom of the Waveform window. You can zoom out, or zoom fit to view more of the time line, reposition the cursor as needed, and then zoom in for greater detail.

13. Select `sineSel` from the list of signals in the Waveform window and use the Next Transition command to move to the specific transition of interest.

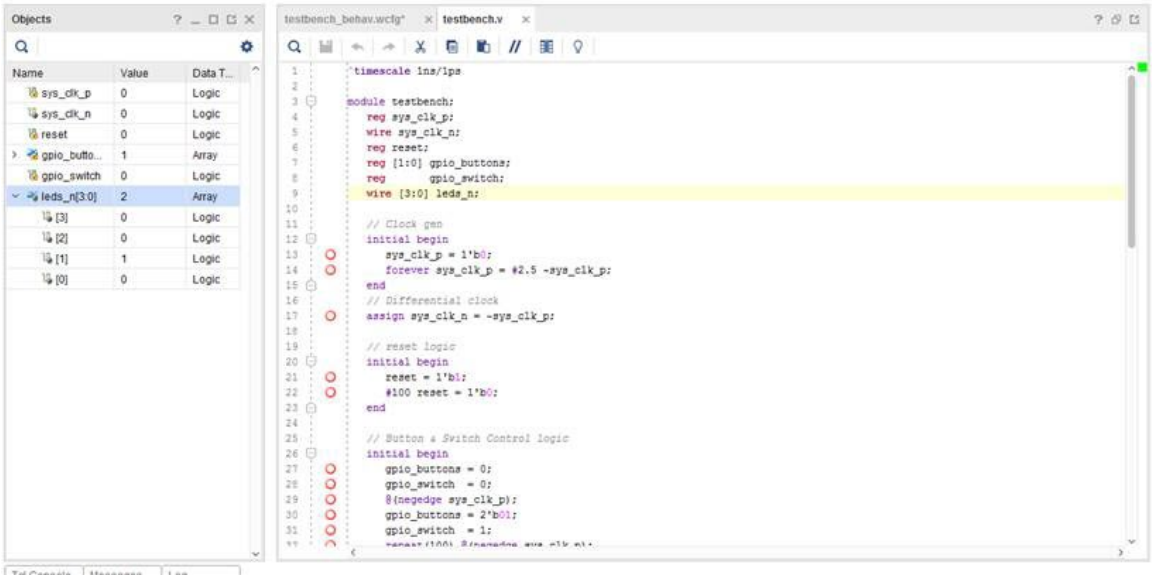
    As shown in above figure, the ruler measures a time period of 3.420 ns as the period that FSM selected the low frequency output.

Step 9: Debugging with Breakpoints

You have examined the design using cursors, markers, and multiple Waveform windows. Now you use Vivado® simulator debugging features, such as breakpoints, and line stepping, to debug the design and identify the cause of the incorrect output.

1. First, open the tutorial design test bench to learn how the simulator generates each design input.
2. Open the `testbench.v` file by double-clicking the file in the Sources window, if it is not already open.

   The source file opens in the Vivado IDE Text Editor, as shown in the following figure.



***Note*:** You can also select **File → Text Editor → Open File** from the main menu, or Open File from the popup menu in the Sources window. You can also select an appropriate design object in the Scopes window or Objects window, right-click and select Go to Source Code.

Using Breakpoints

A breakpoint is a user-determined stopping point in the source code used for debugging the design. When simulating a design with set breakpoints, simulation of the design stops at each breakpoint to verify the design behavior. After the simulation stops, an indicator shows in the text editor next to the line in the source file where the breakpoint was set, so you can compare the Wave window results with a particular event in the HDL source.

You use breakpoints to debug the error with the low frequency signal output that you previously observed. The erroneous `sine[19:0]` output is driven from the `sineGen` VHDL block. Start your debugging with this block.

1. Select the **U_SINEGEN** scope in the Scopes window to list the objects of that scope in the Objects window.
2. In the Objects window, right-click **sine[19:0]** and use **Go to Source Code** to open the `sinegen.vhd` source file in the Text Editor.

   TIP: *If you do not see the `sine[19:0]` signal in the Objects window, make sure that the filters at the top of the Objects window are set properly to include Output objects.*

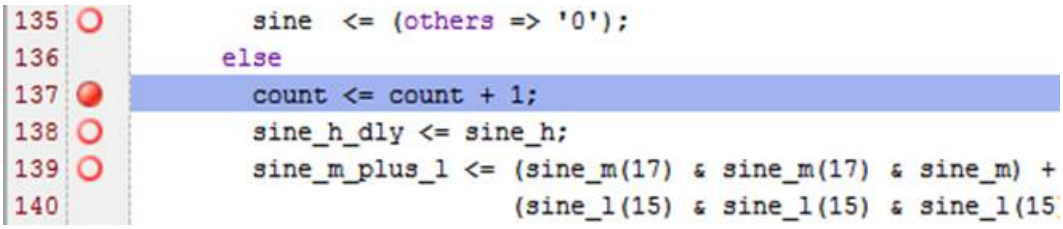
   Looking through the HDL code, the `clk`, `reset`, and `sel` inputs are correct as expected. Set your first breakpoint after the `reset` asserts low at line 137.
3. Scroll to line 137 in the file.

Add a breakpoint at line 137 in `sinegen.vhd`. Note that the breakpoint can be set only on the executable lines. The Vivado® simulator marks the executable lines with an empty red circle , on the left hand margin of the Text Editor, beside the line numbers.

Setting a breakpoint causes the simulator to stop at that point, every time the simulator processes that code, or every time the counter is incremented by one.

4. Click the red circle in the left margin, to set a breakpoint, as shown in the following figure.

   Observe that the empty circle becomes a red dot to indicate that a breakpoint is set on this line. Clicking on the red dot removes the breakpoint and reverts it to the empty circle .



   ***Note*:** To delete all breakpoints in the file, right-click on one of the breakpoints and select Delete All Breakpoints.

   Debugging in the Vivado simulator, with breakpoints and line stepping, works best when you can view the Tcl Console, the Waveform window, and the HDL source file at the same time, as shown in the following figure.

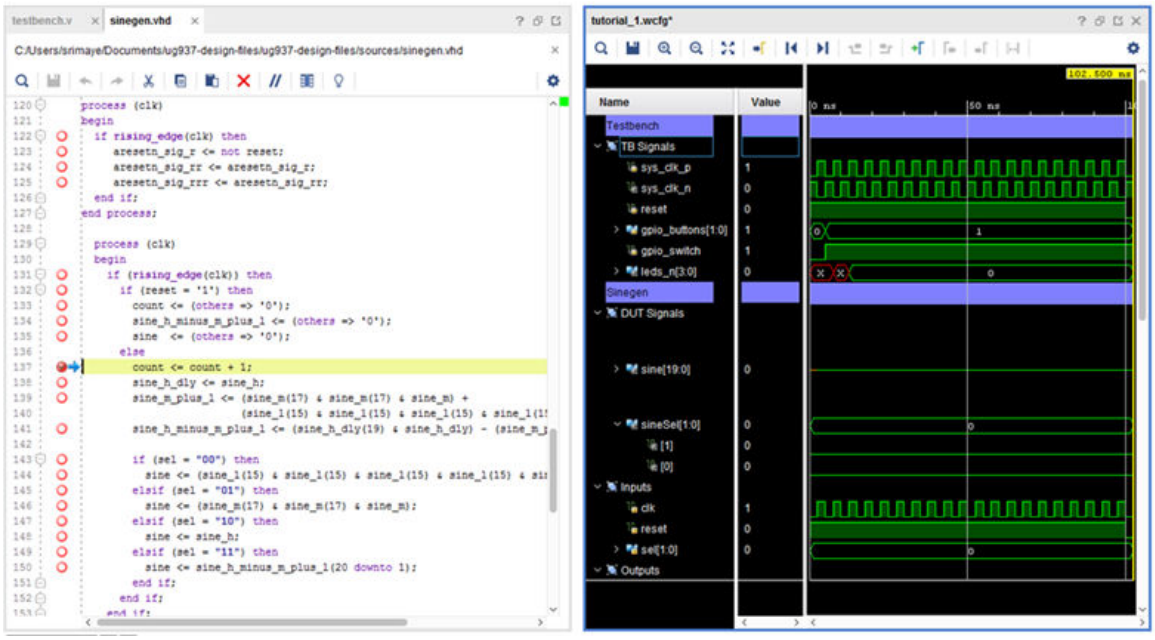
5. Resize the windows, and use the window Float command or the New Vertical Group command to arrange the various windows so that you can see them all.

6. Click the **Restart** button to restart the simulation from time 0.

7. Run the simulation by clicking the **Run All button** .

   The simulation runs to time 102.5 ns, or near the start of first counting, and stops at the breakpoint at line 137. The focus within the Vivado IDE changes to the Text Editor, where it shows the breakpoint indicator and highlights the line.

   A message also displays in the Tcl console to indicate that the simulator has stopped at a specific time, displayed in picoseconds, indicating the line of source code last executed by the simulator.



TIP: *When you have arranged windows to perform a specific task, such as simulation debug in this case, you can save the view layout to reuse it when needed. Use the Layout > Save Layout As command from the main menu to save view layouts. See the Vivado Design Suite User Guide: Using the Vivado IDE (UG893) for more information on arranging windows and using view layouts.*

8. Continue the simulation by clicking the Run All button.

   The simulation stops again at the breakpoint. Take a moment to examine the values in the Waveform window. Notice that the `sine[19:0]` signals in the Outputs group are uninitialized, as are the `sine_1[15:0]` signals in the Internal signals group.

9. In the Text Editor, add another breakpoint at line 144 of the `sinegen.vhd` source file.

   This line of code runs when the value of `sel` is 0. This code assigns, with bit extension, the low frequency signal, `sine_1`, to the output, sine.

10. In the Waveform window, select `sine_1[15:0]` in the Internal Signals group, and holding Ctrl, select `sine[19:0]` in the Outputs group.

    These selected signals are highlighted in the Waveform window, making them easier for you to monitor.

11. Run the simulation by clicking the Run All button.

    Once again, the simulation stops at the breakpoint, this time at line 144.

Stepping Through Source Code

Another useful Vivado® simulator debug tool is the Line Stepping feature. With line stepping, you can run the simulator one-simulation unit (line, process, task) at a time. This is helpful if you are interested in learning how each line of your source code affects the results in simulation.

Step through the source code line-by-line and examine how the low frequency wave is selected, and whether the DDS compiler output is correct.

1. On the Vivado simulator toolbar menu, click the **Step** button  .

   The simulation steps forward to the next executable line, in this case in another source file. The `fsm.vdh` file is opened in the Text Editor. You may need to relocate the Text Editor to let you see all the windows as previously arranged.

   Note: You can also type the `step` command at the Tcl prompt.

2. Continue to Step through the design, until the code returns to line 144 of `sinegen.vhd`.

   You have stepped through one complete cycle of the circuit. Notice in the Waveform window that while `sel` is 0, signal `sine_1` is assigned as a low frequency sine wave to the output sine. Also, notice that `sine_1` remains uninitialized.

3. For debug purposes, initialize the value of `sine_1` by entering the following add_force command in the Tcl console:

   ```
   add_force /testbench/dut/U_SINEGEN/sine_1 0110011011001010
   ```

   This command forces the value of `sine_1` into a specific known condition, and can provide a repeating set of values to exercise the signal more vigorously, if needed. Refer to the *Vivado Design Suite User Guide: Logic Simulation* (UG900) for more information on using `add_force`.

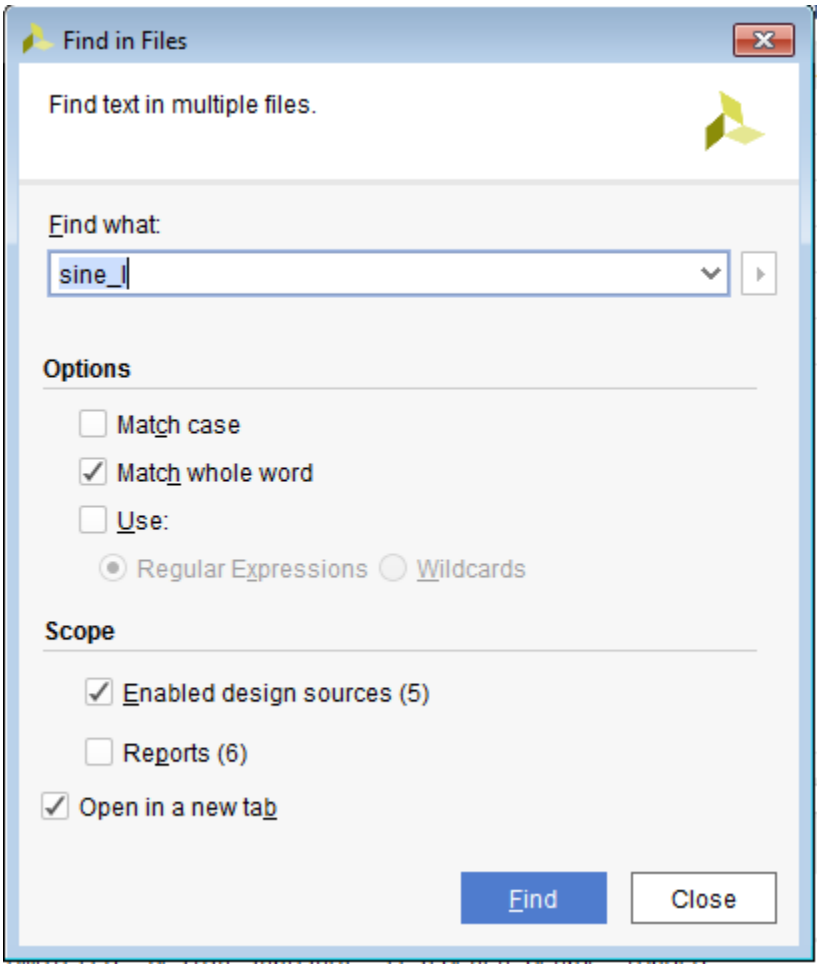
4. Continue the simulation by clicking the **Run All** button  a few more times.

   In the Waveform window, notice that the value of `sine_1[15:0]` is now set to the value specified by the add_force command, and this value is assigned to the output signal `sine[19:0]` since the value of `sel` is still 0.

   Trace the `sine_1` signal in the HDL source files, and identify the input for `sine_1`.

5. In the Text Editor, right-click to open the popup menu, and select the **Find in files** option to search for `sine_1`.

6. Select the **Match whole word** and **Enabled design sources** checkboxes, as shown in the following figure, and click **Find**.



The Find in Files results display at the bottom of the Vivado IDE, with all occurrences of `sine_l` found in the `sinegen.vhd` file.

7. Expand the Find in Files results to view the results in the `sinegen.vhd` file.

   The second result, on line 111, identifies a problem with the design. At line 111 in the `sinegen.vhd` file, the `m_axis_data_tdata_sine_low` signal is assigned to `sine_l`. Since line 111 is commented out, the `sine_l` signal is not connected to the low frequency DDS compiler output, or any other input.

8. Uncomment line 111 in the `sinegen.vhd` file, and click the **Save File** button .

9. In the Tcl Console, remove the force on `sine_l`: `remove_forces -all`

Step 10: Relaunch Simulation

By using breakpoints and line stepping, you identified the problem with the low frequency output of the design and corrected it.

Because you modified the source files associated with the design, you must recompile the HDL source and build new simulation snapshot. Do not just restart the simulation at time 0 in this case but rebuild the simulation from scratch.

1. In `sinegen.vhd`, select one of the breakpoints, right-click and select **Delete All Breakpoints**.
2. Click the Relaunch button on the main toolbar menu.

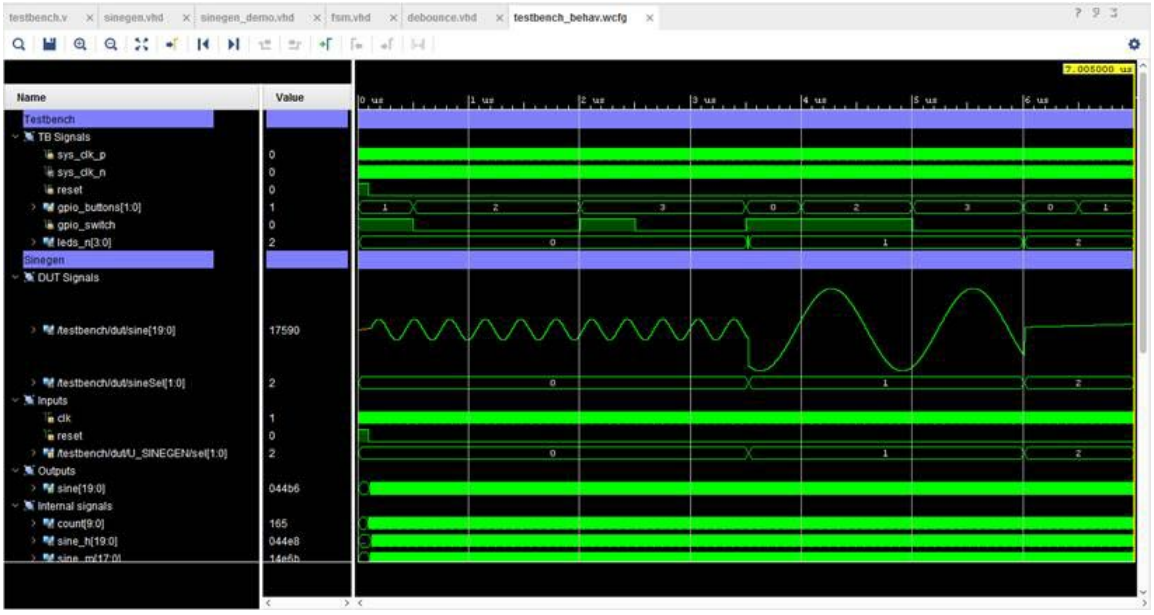
   Note: If prompted to save the Wave Config file, click **yes**.

   The Vivado® simulator recompiles the source files with `xelab`, and re-creates the simulation snapshot. Now you are ready to simulate with the corrected design files. The relaunch button will be active only after one successful run of Vivado Simulator using launch_simulation. If you run the simulation in a Batch/Scripted mode, the relaunch button would be greyed out.

3. Click the **Run All** button to run the simulation.

   Observe the `sine[19:0]`, the analog signal in the waveform configuration. The low frequency sine wave looks as expected. The Tcl console results are:

```
[@3518000] LEDS_n = 0100
[@3523000] LEDS_n = 0001
[@3523000] LEDS_n = 0001
[@6008000] LEDS_n = 0101
[@6013000] LEDS_n = 0010
[@6013000] LEDS_n = 0010
$finish called at time : 7005 ns : File "ug937/sim/testbench.v" Line 63
```



Conclusion

After reviewing the simulation results, you may close the simulation, and close the project. This completes Lab #2. Up to this point in the tutorial, between Lab #1 and Lab #2, you have:

- Run the Vivado® simulator using the Project Mode flow in Vivado® IDE
- Created a project, added source files, and added IP
- Added a simulation-only file (`testbench.v`)
- Set simulation properties and launched behavioral simulation
- Added signals to the Waveform window
- Configured and saved the Waveform Configuration file
- Debugged the design bug using breakpoints and line stepping.
- Corrected an error, re-launched simulation, and verified the design

In Lab # 3, you will examine the Vivado simulator batch mode.

Lab 3: Running Simulation in Batch Mode

You can use the Vivado® simulator Non-Project Mode flow to simulate your design without setting up a project in Vivado® Integrated Design Environment (IDE).

In this flow, you:

- Prepare the simulation project manually by creating a Vivado simulator project script.
- Create a simulation snapshot file using the Vivado simulator xelab utility.
- Start the Vivado simulator GUI by running the xsim command with the resulting snapshot.

Step 1: Preparing the Simulation

The Vivado® simulator Non-Project Mode flow lets you simulate your design without setting up a project in the Vivado® IDE.

You can compile the HDL files in a design, and create a simulation snapshot by either:

- Creating a Vivado simulator project script, specifying all HDL files to be compiled, and using the xelab command to create a simulation snapshot, or
- Using specific Vivado simulator parser commands, xvlog and xvhdl, to parse individual source files and write the parsed files into an HDL library on disk, and then using xelab to create a simulation snapshot from the parsed files

Creating the Vivado Simulator Project File

A Vivado® simulator project script specifies design source files and libraries to parse and compile for simulation. This method is useful to create a simulation project script that can be run repeatedly over the course of project development.

The format for a Vivado simulator project script (prj file) is as follows:

```
verilog | vhdl| sv <library_name> {<file_name>.v|.vhd|.sv
```

where,

- `verilog | vhdl | sv` specifies whether the design source is a Verilog, VHDL, or SV file.
- `<library_name>` specifies the library into which you may compile the source file. If unspecified, the default library for compilation is work.
- `<file_name>.v|.vhd|.sv` specifies the name of the design source file to compile.

IMPORTANT! *While you can specify one or more Verilog source files on a single command line, you can only specify one VHDL source on a single command line.*

In this step, you build a Vivado simulator project script by editing an existing project script to add missing source files. The command lines for the project script should be constructed using the syntax described above.

1. Browse to the `<Extract_Dir>/scripts` folder.
2. Open the `simulate_xsim.prj` project script with a text editor.
3. Add the following commands to the project script:

```
vhdl xil_defaultlib "../sources/sinegen.vhd"
vhdl xil_defaultlib "../sources/debounce.vhd"
vhdl xil_defaultlib "../sources/fsm.vhd"
vhdl xil_defaultlib "../sources/sinegen_demo.vhd"
verilog xil_defaultlib "../sim/testbench.v"
```

4. Save and close the file.

You do not need to list the sources based on any specific order of dependency. The `xelab` command resolves the order of dependencies, and automatically processes the files accordingly.

TIP: *For your reference, a completed version of the tutorial files can be found in the `ug937-design-files/completed` folder.*

Manually Parsing Design Files

As an alternative to creating a Vivado® simulator project script, you can compile individual design source files directly from the command line using the `xvlog` or `xvhdl` commands to parse the design sources and write them to an HDL library. You could use this method for simple simulation runs, or to define a shell script and makefile compilation flow.

Parse individual or multiple Verilog files using the `xvlog` command with the following syntax format:

```
xvlog [options] <verilog_file | list_of_files>
```

Parse individual VHDL files using the `xvhdl` command with the following syntax format:

```
xvhdl [options] <VHDL_file>
```

For a complete list of available `xvlog` and `xvhdl` command options, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900). The parse_standalone.bat file in `<Extract_Dir>/scripts` or `<Extract_Dir>/completed` provide examples of running `xvlog` and `xvhdl` directly.

Step 2: Building the Simulation Snapshot

In this step, you use the `xelab` command on the project script you previously edited (`simulate_xsim.prj`) to elaborate, compile, and link all the sources for the design. The `xelab` utility creates a simulation snapshot that lets you to simulate the design in the Vivado® simulator.

The typical `xelab` command syntax is:

```
xelab -prj <project_file> -s <simulation snapshot> <library>.<top_unit>
```

where,

- `-prj <project_file>`: Specifies a Vivado simulation project script to use for input.
- `-s <simulation_snapshot>`: Specifies the name of the output simulation snapshot.
- `<library>.<top_unit>`: Specifies the library and top-level module of the design.

Running xelab

In this step, you use the `xelab` command with the project file completed in Step 1 to elaborate, compile, and link all the design sources to create the simulation snapshot. To run the `xelab` command, open and configure a command window.

1. On Windows, open a Command Prompt window. On Linux, simply skip to the next step.
2. Change directory to the Xilinx installation area, and run settings64.bat as needed to setup the Xilinx tool paths for your computer:

   ```
   cd install_path\Vivado\2021.x\
   ```

   ```
   settings64.bat
   ```

   Note: The `settings64.bat` file configures the path on your computer to run the Vivado ML Editions.

   TIP: *When running the `xelab`, `xsc`, `xsim`, `xvhdl`, or `xvlog` commands in batch files or scripts, it may also be necessary to define the XILINX_VIVADO environment variable to point to the installation hierarchy of the Vivado ML Editions. To set the XILINX_VIVADO variable, you can add one of the following to your script or batch file:*

On Windows -

```
set XILINX_VIVADO=<Vivado_install_area>/Vivado/2021.x
```

On Linux -

```
setenv XILINX_VIVADO <Vivado_install_area>/Vivado/2021.x
```

or

```
export XILINX_VIVADO=<Vivado_install_area>/Vivado/2021.x
```

3. Change directory to the `<Extract_Dir>/scripts` folder.

   The provided `xelab` batch file, `xelab_batch.bat`, is incomplete and you must modify it using the `xelab` syntax as previously described to produce the correct simulation snapshot.

4. Edit the `xelab_batch.bat` file to add the following options:

   - Specify the project file: `-prj simulate_xsim.prj`
   - Specify the output simulation snapshot: `-s run_sineGen`
   - Specify the library and top-level design unit: `xil_defaultlib.testbench`

   For a complete list of available xelab command options, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900).

5. Save and close the batch file.
6. In the command window, run the `xelab_batch.bat` file to compile and create the simulation snapshot.

   ```
   xelab_batch.bat
   ```

7. Examine the `xelab` output as it is transcribed to the Command Prompt window.

   Note: The `xelab` command also writes `xelab.log` file in the directory from which it was run. The log file contains all of the messages and results of the `xelab` command for you to review.

TIP: *You can also use the `xelab` command after the `xvlog` and `xvhdl` commands have parsed the HDL design sources to read the specified simulation libraries. The `xelab` command would be the same as described here, except that it would not require the `-prj` option since there would be no simulation project file.*

Step 3: Manually Simulating the Design

In this step, you launch the Vivado® simulator GUI by running the `xsim` command with the simulation snapshot that you generated using the `xelab` command in Step 2. After you complete this step, you can use the Vivado simulator GUI to explore the design in more detail.

In the same command window that you used for Step #2, type the following command:

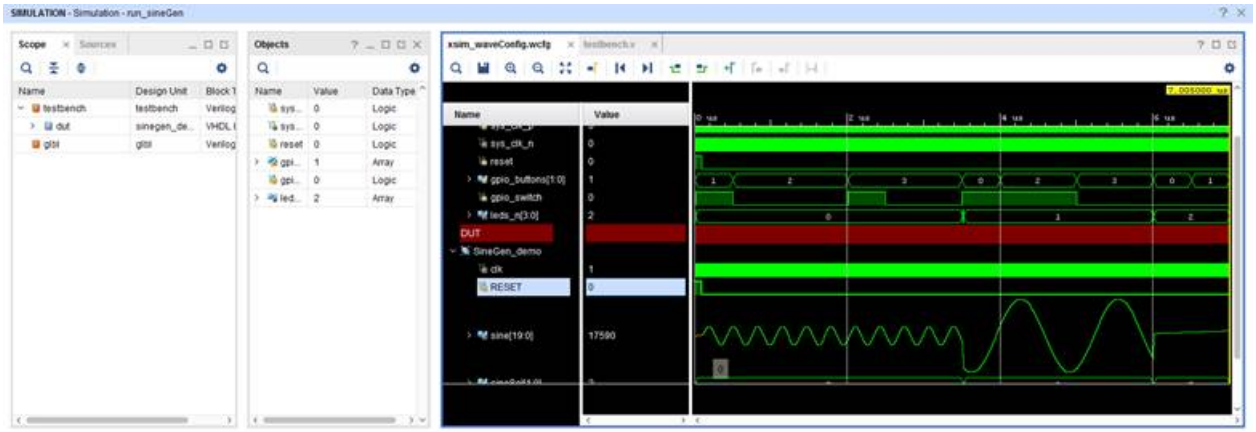
```
xsim run_sineGen -gui -wdb simulate_xsim.wdb -view xsim_waveConfig
```

where:

- `run_sineGen -gui`: Specifies the simulation snapshot that you generated using `xelab`, and launches Vivado simulator in GUI mode.
- `-wdb`: Specifies the file name of the simulation waveform database file to output, or write, upon completion of the simulation run.
- `-view`: Opens the specified waveform configuration file within the Vivado simulator GUI.

Note: You can use the waveform configuration file specified above, or use the `tutorial_1.wcfg` file that you created in Lab #2 of this tutorial.

The Vivado Simulator GUI opens and loads the design (see the following figure). The simulator time remains at 0 ns until you specify a run time. Run the simulation and explore the design.



Conclusion

In this tutorial, you:

- Created a Vivado® IDE project
- Downloaded source files and ran Vivado simulation
- Examined the simulation customization features
- Debugged and fixed a known issue within the source files
- Ran a Vivado simulation in batch mode using the Vivado simulation executable and switch options

Lab 4: System Verilog Feature

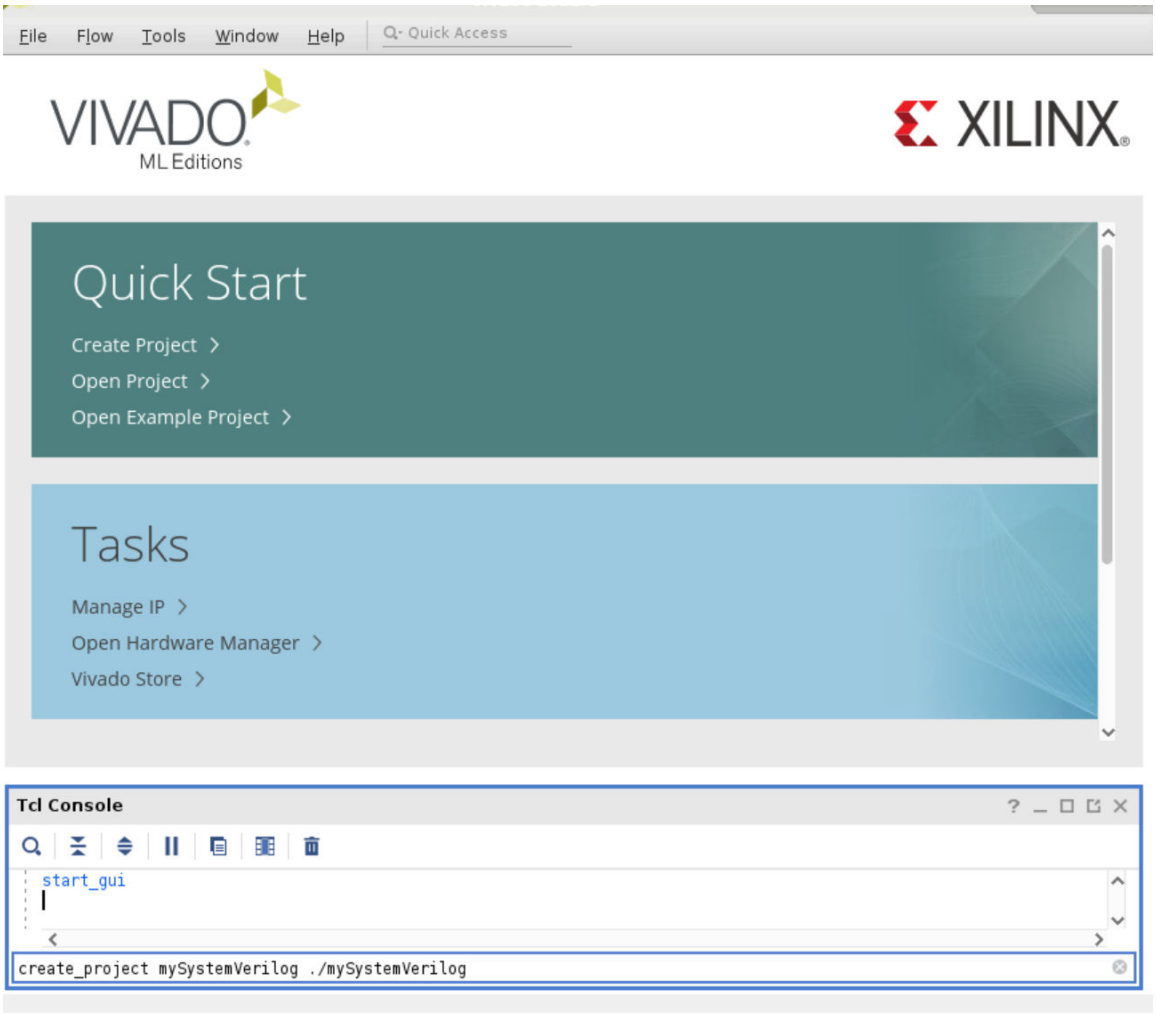
Vivado® simulator now supports synthesizable as well as test bench/verification feature of System Verilog IEEE 1800-2012. In this chapter, you will go through a System Verilog example to learn about different debugging capabilities added in the Vivado simulator. You will use an IP example design provided with Vivado.

Creating an Example Design

You will now generate an AXI-VIP example design.

1. Open Vivado®.
2. Create a project with the name `mySystemVerilog` by invoking the following command in Vivado Tcl console.

```
create_project mySystemVerilog ./mySystemVerilog
```



3. You will create an AXI-VIP example design that includes the following features:
   - Random Constraint
   - Dynamic Types and Class
   - Virtual Interface
   - Assertion
   - Clocking Block
4. Invoke the following commands in Tcl console:
   a. `create_ip -name axi_vip -vendor xilinx.com -library ip -version 1.1 -module_name axi_vip_0`
   b. `open_example_project -force [get_ips axi_vip_0]`

Now you have created an example design for AXI-VIP with the name `axi_vip_0_ex`.

Launching Simulation

You have an example project ready. Next, you will run the behavioral simulation. By default, the simulation runs in a pre-compiled mode where the source code for static IP is not added in the project. Run the simulation in a non-precompiled mode for a better understanding of the feature. Invoke the following commands in Tcl console:

- `set_property sim.use_ip_compiled_libs 0 [current_project]`
- `launch_simulation`

This will run the simulation for 1000 ns.

Debugging Using Vivado Simulator

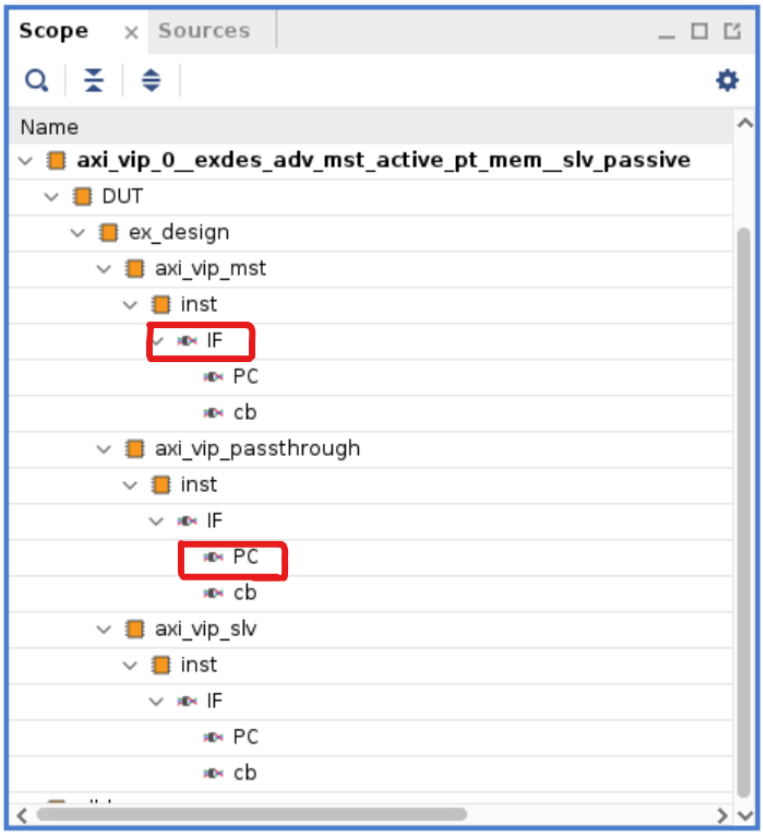
Vivado® simulator supports System Verilog feature. In this exercise, you will explore the System Verilog feature using the following:

- Scope Window
- Object Window
- Tcl Console

Scope Window

System Verilog has a building block called interface. It differs in functionality when you compare it with the module.

1. On the Scope window, click **Expand All** button  .
2. You can now view the IF and PC interface instances. The IF and PC interface instance icons are different than the module icons.



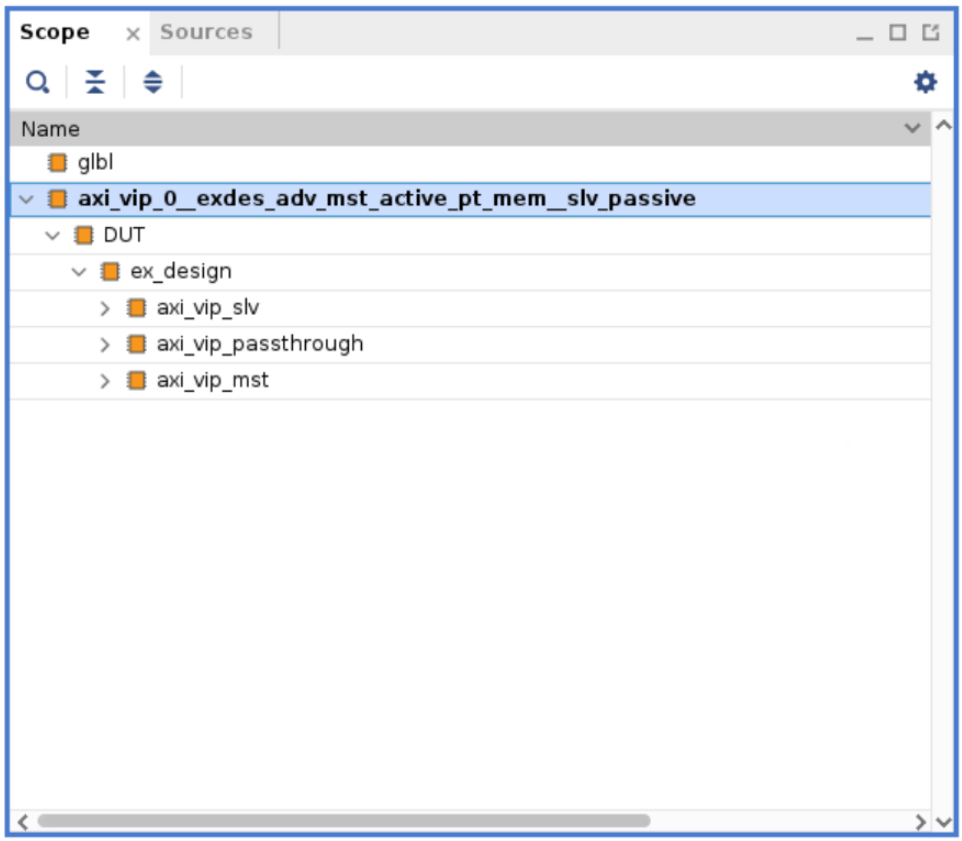
3. Right-click **IF** and select **Go to Source Code**. It will point to an interface definition.

Object Window

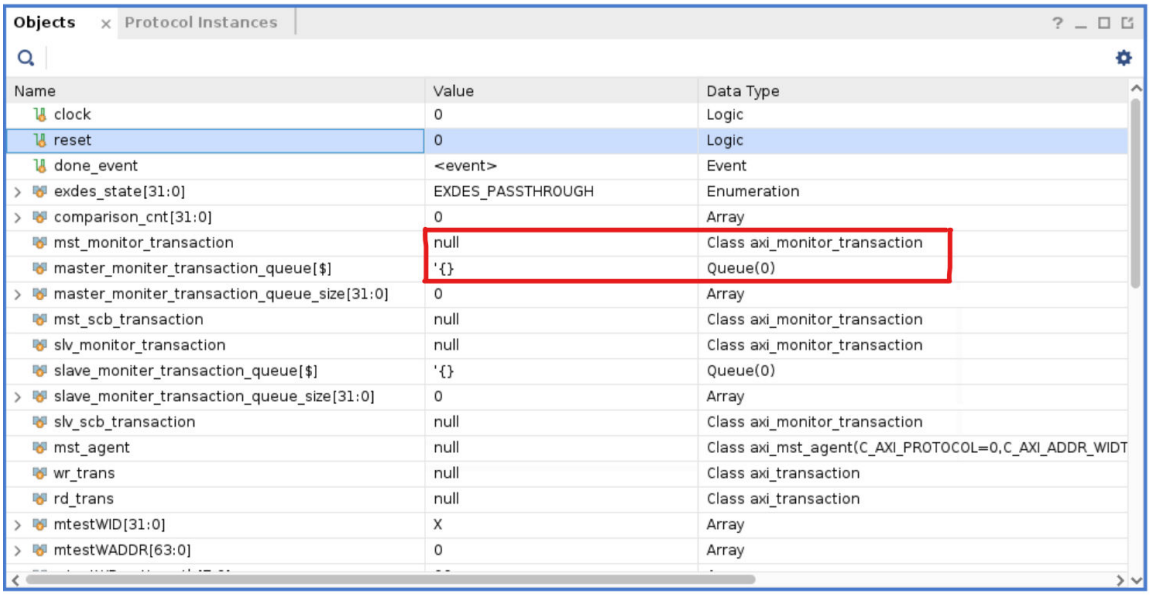
In System Verilog, all the net/variables are static type. They exist throughout the simulation. In System Verilog, dynamic type is a new type along with static type. Class, Queue, and Associative Array are some examples of dynamic type.

Unlike static type variables (int a; wire [7:0] b;), dynamic type variables do not have a fixed size throughout the simulation. Variables keep changing during run-time. Through Object window, you can view the value of a dynamic type variable during the simulation.

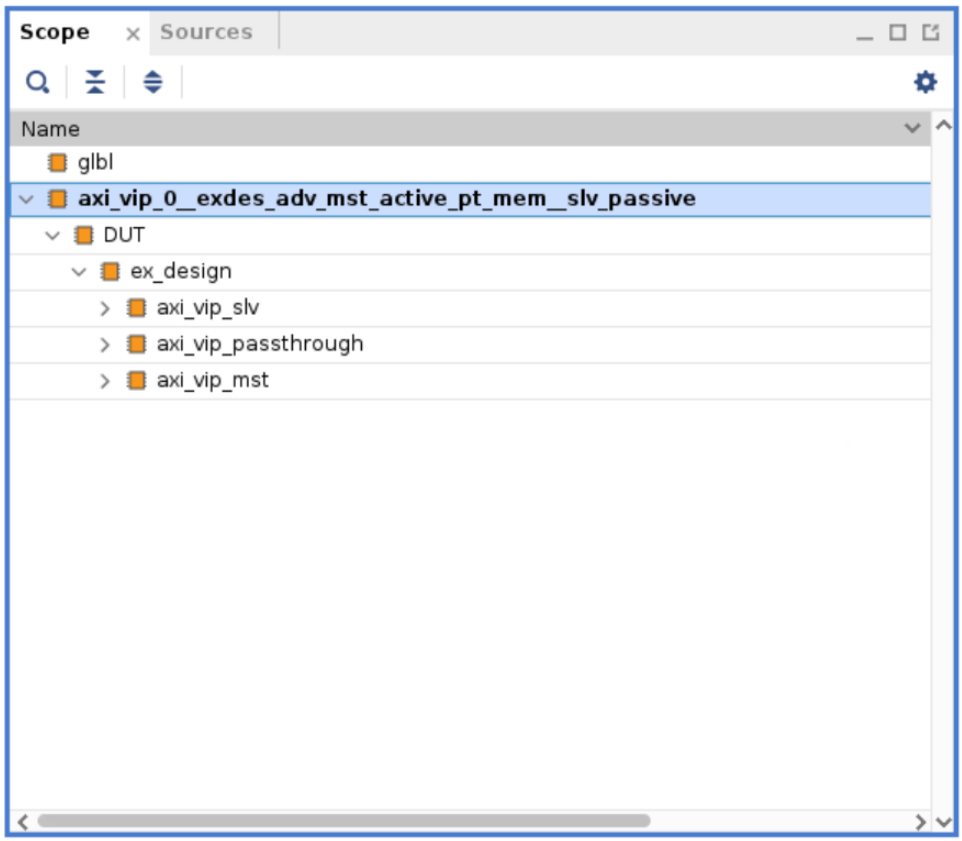
1. Click **Restart** button
2. From the Scope window, select scope **axi_vip_0__exdes_adv_mst_active_pt_mem__slv_passive**.



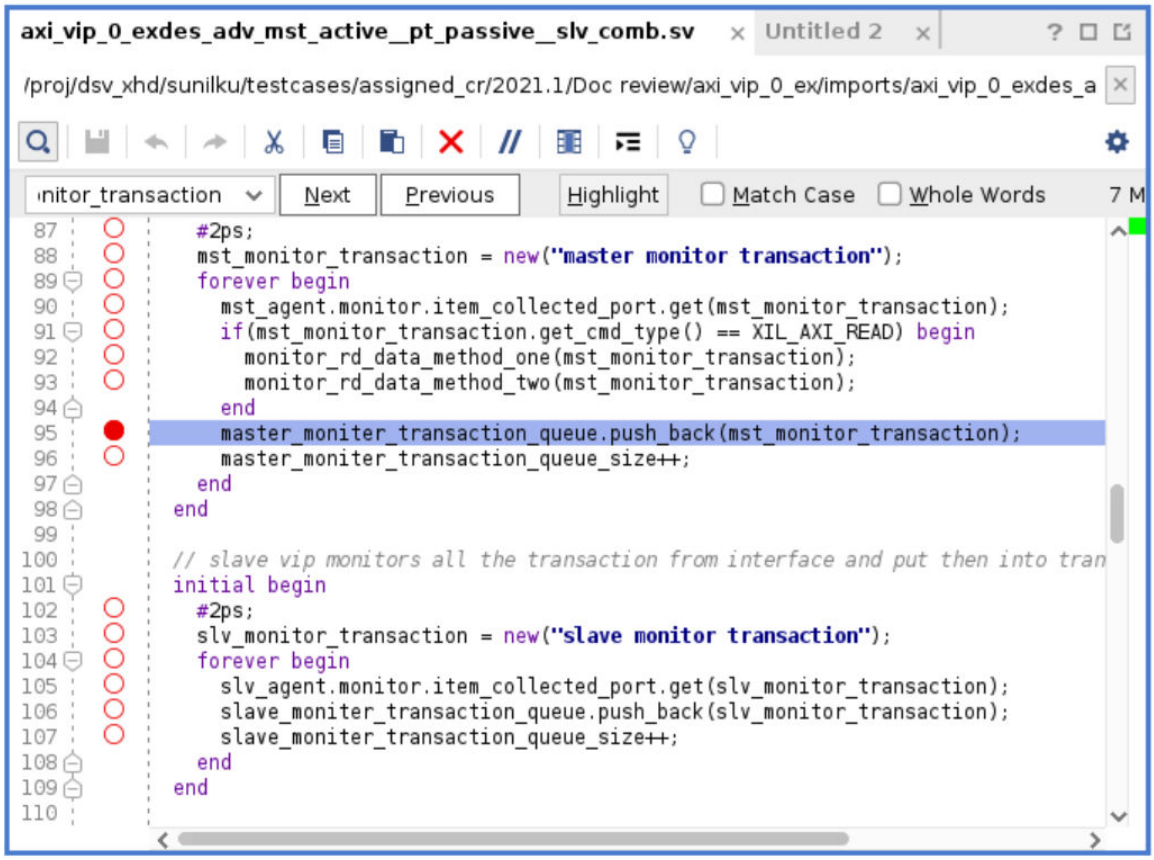
3. Maximize the Objects window. As the simulation is yet to start, observe the Queue and Class dynamic type in the Data Type column. The Value for Queue appears empty while for class it appears null.



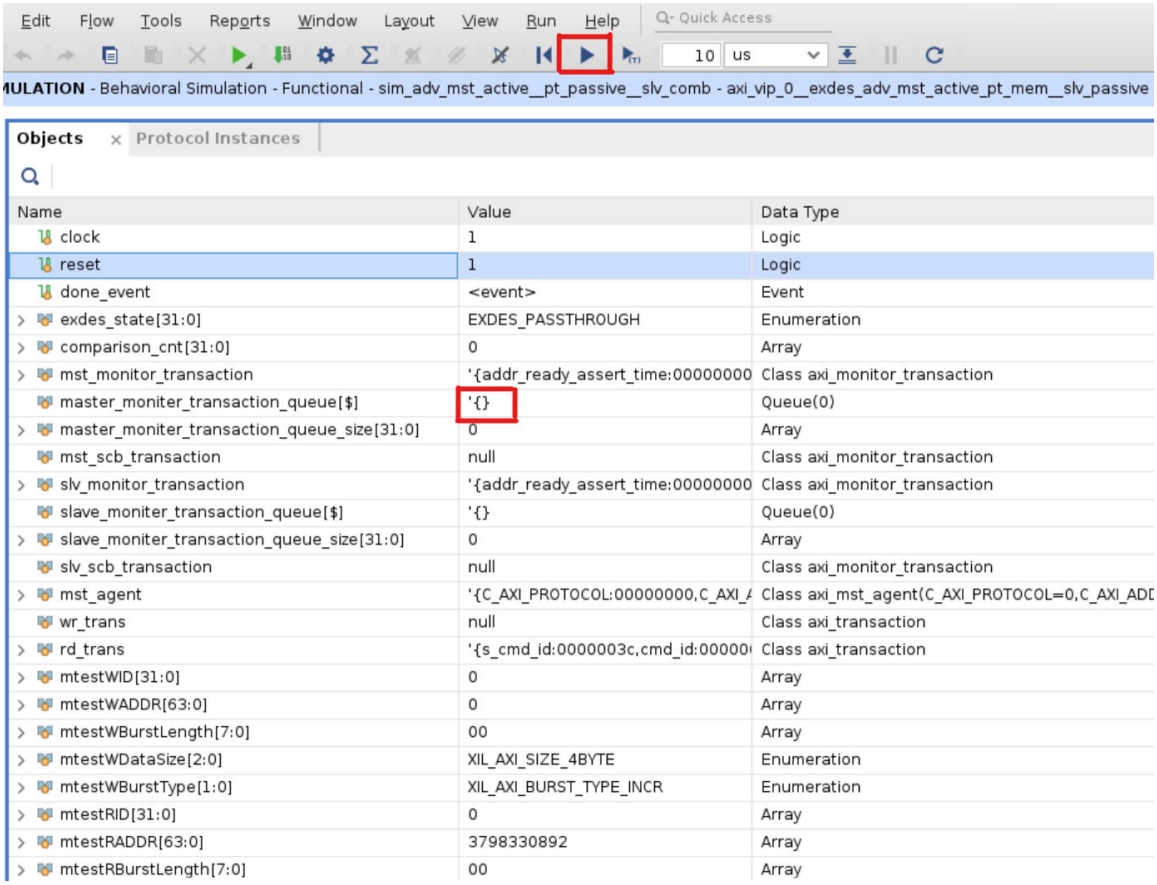
4. On Scope window, double-click **axi_vip_0__exdes_adv_mst_active_pt_mem__slv_passive** to see the text file.



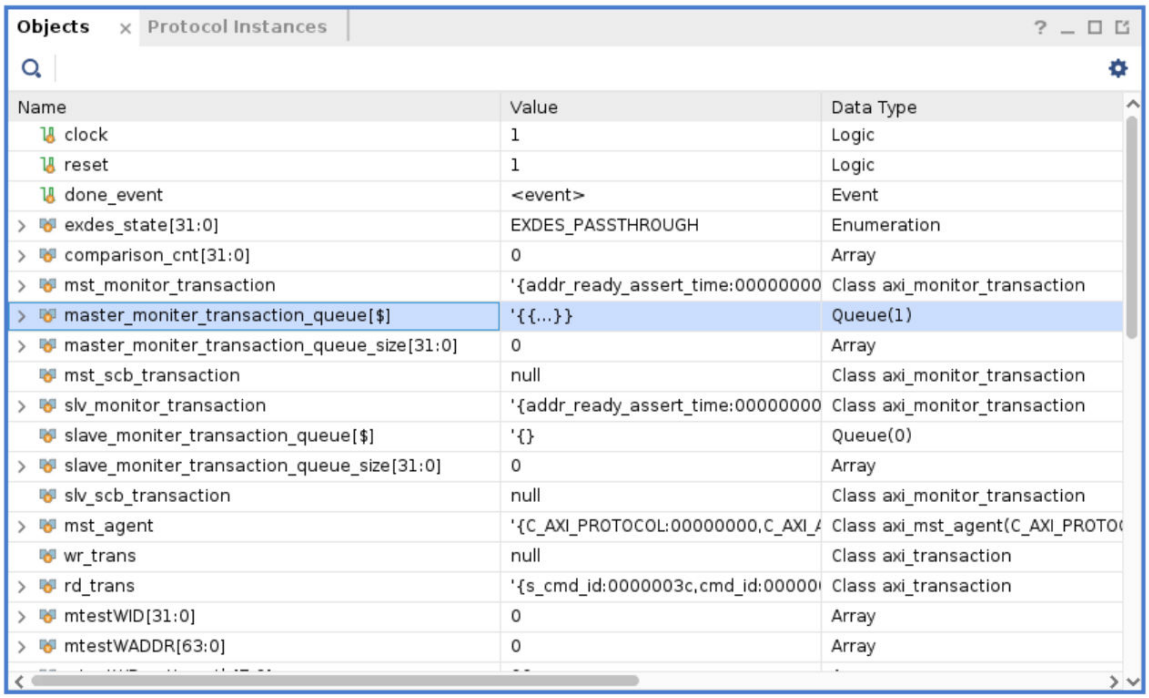
5. On the text editor window, click the **circle** on line number 95 to add a break point.



6. Click **Run All** button, the simulation will stop at line number 95. On the Object window `master_monitor_transaction_queue` value appears empty.



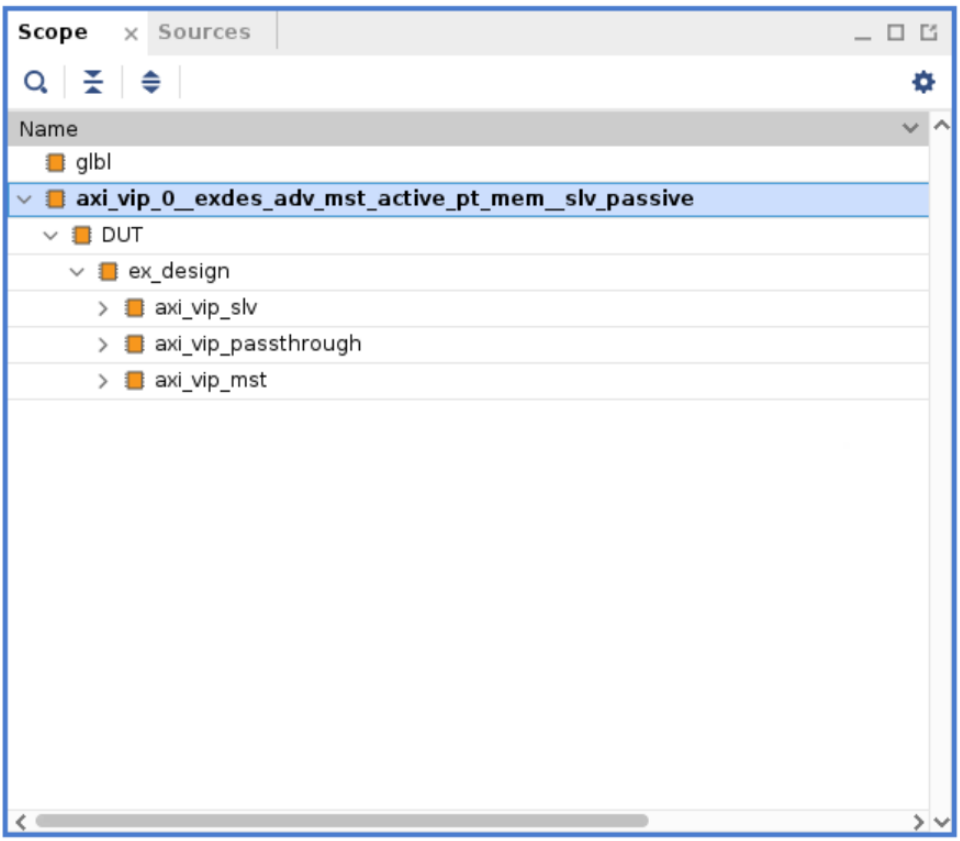
7. On the Vivado® simulator toolbar menu, click the **Step** button . It executes the current statement that is on line number 95 where the simulation is currently waiting. At this statement, you are pushing an element after the execution. Your queue will be populated with a single element.
8. In the Object window, the `master_monitor_transaction_queue` value is populated. This way you can view the value of any dynamic type on the Object window.



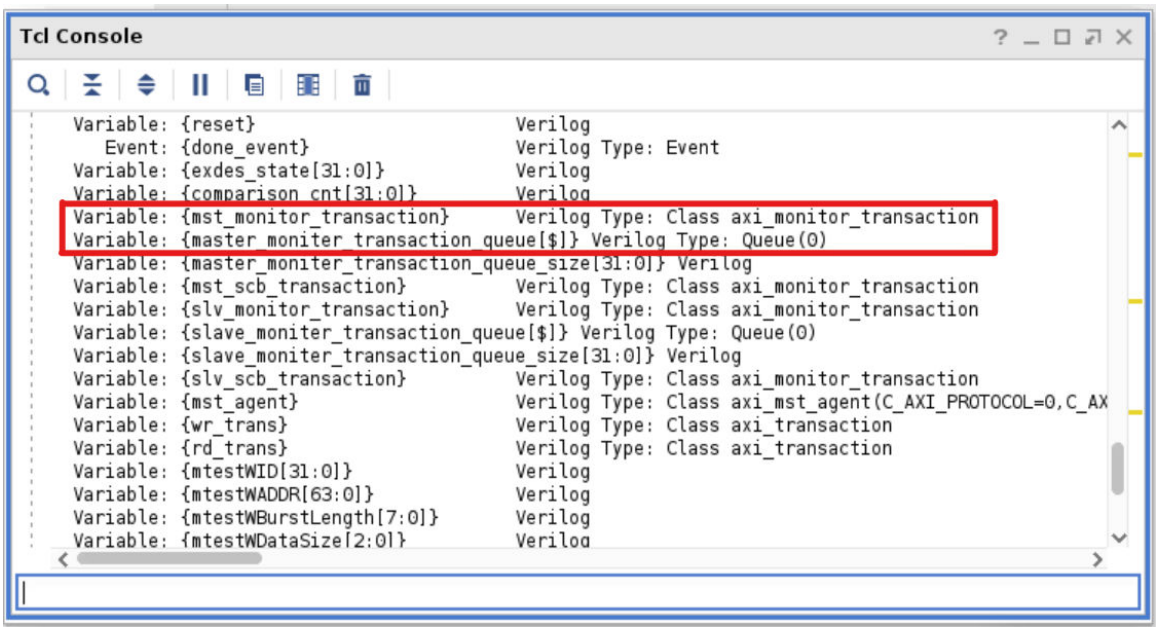
Tcl Console

Like Objects window, you can view the value of any dynamic type variable from Tcl Console as well.

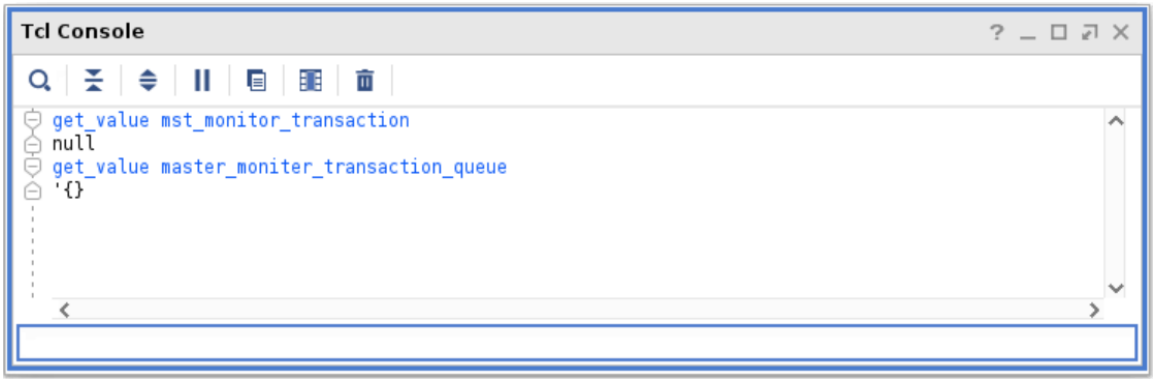
1. Click **Restart** button .
2. From the Scope window, select scope **axi_vip_0__exdes_adv_mst_active_pt_mem__slv_passive** present under the top module.



3. In Tcl console, invoke the `report_objects` command to display all objects present in the selected scope. Also, the Queue and Class appear as object type.



4. Invoke the `get_value` command to find the current value of an object. The get value of `mst_monitor_transaction` is returning null as its Class type while empty parenthesis appears for `master_monitor_transaction_queue` which is a Queue type.



5. Click **Run All** button . The simulation stops at the line where you have added the break point.
6. Invoke `get_value master_moniter_transaction_queue` command and notice that it is still empty.
7. On the Vivado® simulator toolbar menu, click the **Step** button . It executes the current statement that is on line number 95 where the simulation is currently waiting. At this statement, you are pushing an element after the execution. Your queue will be populated with a single element.
8. Invoke the `get_value master_moniter_transaction_queue` command and notice 1 entry in the Queue. Like the Objects window, you can read the value of any dynamic type variable in Tcl Console.

Functional Coverage

Functional coverage is a user defined metric that measures the extent to which the design specification, as enumerated by features in the test plan, was exercised. It can be used to measure whether interesting scenarios, corner cases, specification invariants, or other applicable design conditions—captured as features of the test plan—have been observed, validated, and tested.

The Vivado® simulator supports functional coverage. If your design contains any functional coverage statement, the tool will generate a database (coverage database). To view coverage database, Vivado® simulator provides a utility named as xcrg (Xilinx Coverage Report Generator). Refer to the *Vivado Design Suite User Guide: Logic Simulation* (UG900) for more information on functional coverage and xcrg.

In the present example design, you will add a functional coverage code to view the utility of xcrg.

Cover Group Declaration

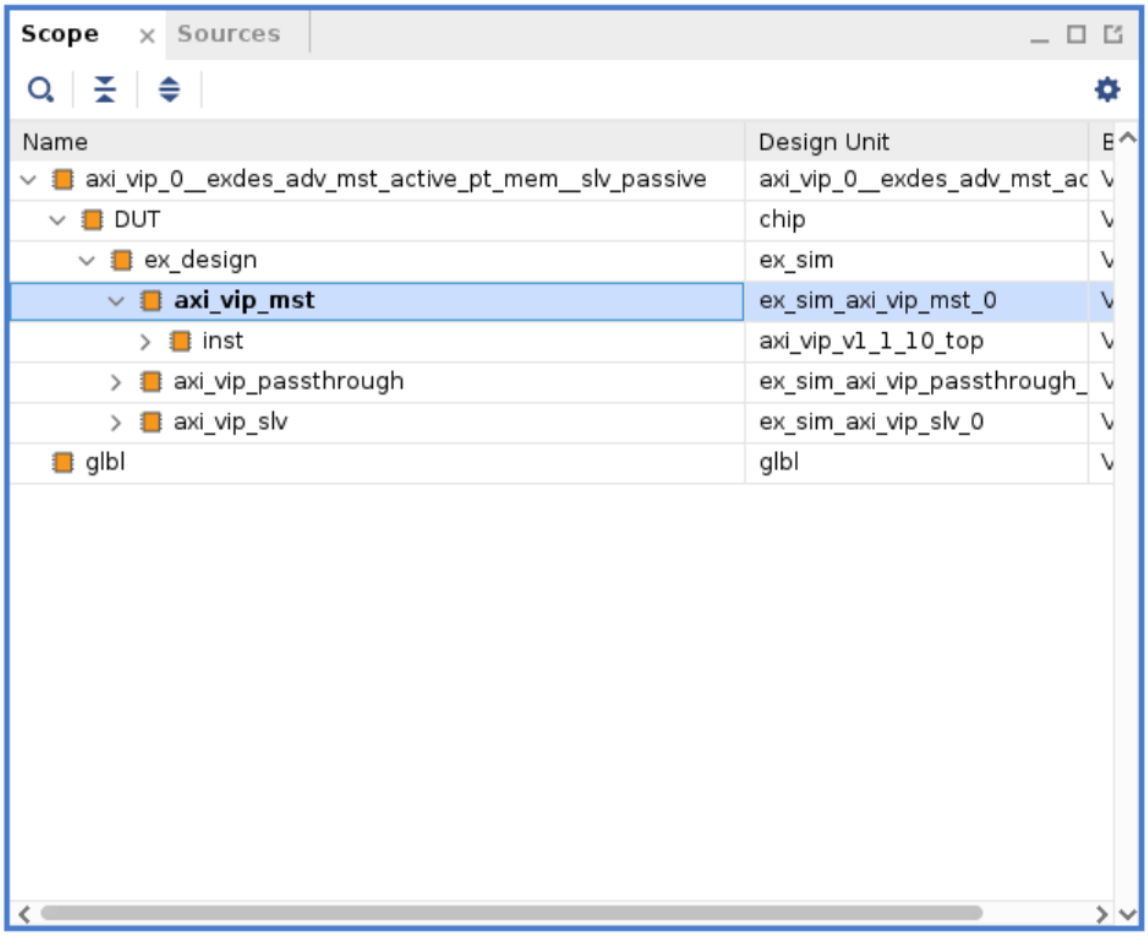
```
covergroup my_cover_group @(posedge aclk);
 m_axi_awlen_cp : coverpoint m_axi_awlen;
 m_axi_awcache_cp : coverpoint m_axi_awcache {
        option.comment = "cp with transition bins";
        bins a1 = (1=>2);
        bins b1 = (1,3=>4,5);
        bins b2[] = (1,3=>4,5);
        bins b3 = (1=>2), ([4:6] => 11,12);
        }
m_axi_bresp_cp : coverpoint m_axi_bresp;
endgroup
my_cover_group obj1 = new();
```

In this example, you are declaring a covergroup named as my_cover_group and the sampling event as posedge aclk. This covergroup contains three coverpoints. You will add the cover group declaration code in the example design.

1. In Tcl console invoke the following command:

   ```
   current_scope /axi_vip_0__exdes_adv_mst_active_pt_mem__slv_passive/DUT/
   ex_design/axi_vip_mst
   ```

2. Double-click **axi_vip_mst** scope to see the source code.



3. The following file path will appear in the text editor, then add the cover group declaration before endmodule and save it.

```
axi_vip_0_ex/axi_vip_0_ex.ip_user_files/bd/ex_sim/ip/
ex_sim_axi_vip_mst_0/sim/ex_sim_axi_vip_mst_0.sv
```

```
    .m_axi_rvalid(m_axi_rvalid),
    .m_axi_rready(m_axi_rready)
  );
covergroup my_cover_group @(posedge aclk);
m_axi_awlen_cp : coverpoint m_axi_awlen;
m_axi_awcache_cp : coverpoint m_axi_awcache {
        option.comment = "cp with transition bins";
        bins a1 = (1=>2);
        bins b1 = (1,3=>4,5);
        bins b2[] = (1,3=>4,5);
        bins b3 = (1=>2), ([4:6] => 11,12);
        }
 m_axi_bresp_cp : coverpoint m_axi_bresp;
endgroup

my_cover_group obj1 = new();

endmodule
```

4. In Tcl console, invoke`close_sim` command to close the simulation running previously.
5. In Tcl console, invoke `reset_simulation` command to clean the simulation directory.
6. In Tcl console, invoke `launch_simulation` command to run the simulation.
7. Click **Run All** button  .

The simulation will stop after reaching $finish statement. The Vivado® simulator has generated the coverage database at the following location with the name (default name) `xsim.covdb`:

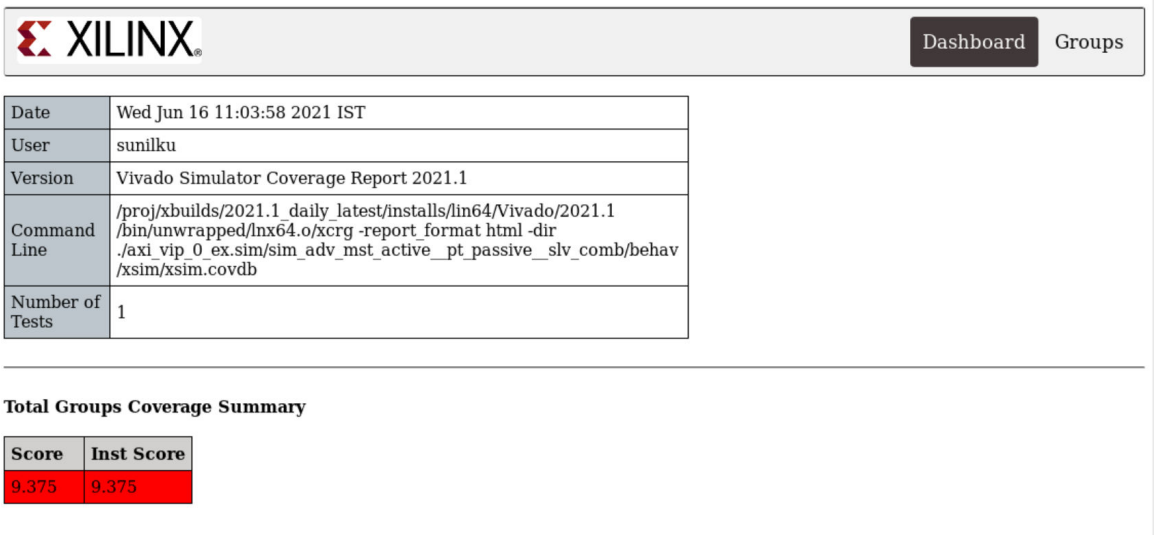
```
./axi_vip_0_ex.sim/sim_adv_mst_active__pt_passive__slv_comb/behav/xsim/
```

Invoke the following command to generate a report:

```
xcrg -report_format html -dir ./axi_vip_0_ex.sim/
sim_adv_mst_active__pt_passive__slv_comb/behav/xsim/xsim.covdb/
```

This will generate a directory with the name `xcrg_report`, it contains a `.html` report. The following is the description of an example report:

1. Open `dashboard.html` file. The file contains details such as command, version, date and coverage summary that shows only 9.375% of total bins are covered.



XILINX

Dashboard Groups

Date	Wed Jun 16 11:03:58 2021 IST
User	sunilku
Version	Vivado Simulator Coverage Report 2021.1
Command Line	/proj/xbuilds/2021.1_daily_latest/installs/lin64/Vivado/2021.1 /bin/unwrapped/lnx64.o/xcrg -report_format html -dir ./axi_vip_0_ex.sim/sim_adv_mst_active__pt_passive__slv_comb/behav /xsim/xsim.covdb
Number of Tests	1

Total Groups Coverage Summary

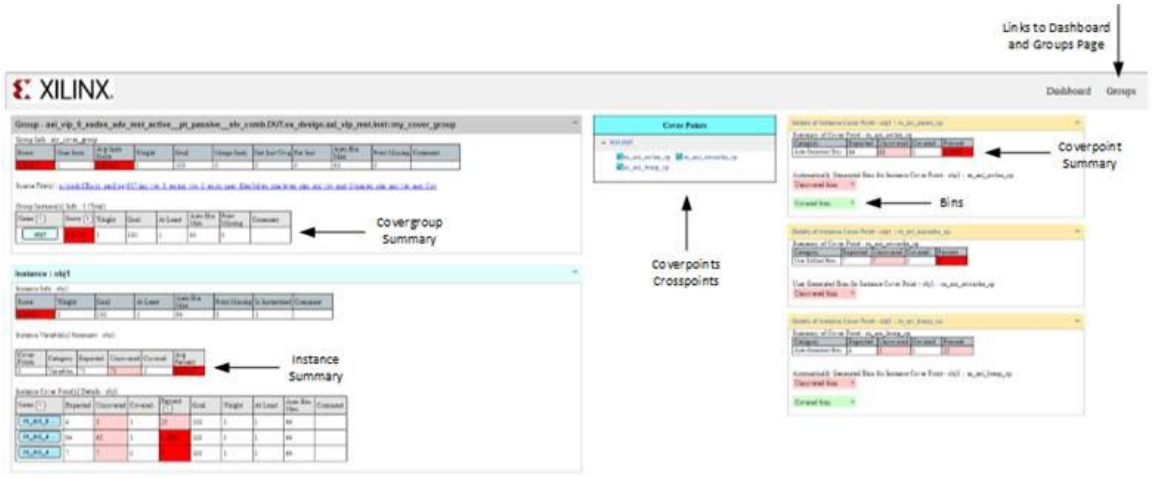
Score	Inst Score
9.375	9.375

2. Click **Groups** button.
3. Click the link under group report.

Total groups in report: 1

Name	Score	Num Instances
axi_vip_0__exdes_adv_mst_active_pt_mem__slv_passive.DUT.ex_design.axi_vip_mst.inst::my_cover_group	9.375	1

4. You will see a detailed report as shown in the following figure:



This way you can view the coverage and change your test bench/seed value to improve the coverage.

Assertion

In System Verilog, you have the following two types of assertion:

- Immediate assertion
- Concurrent Assertion

Immediate Assertion

Evaluated like an expression in 'if' statement.

```
always@(posedge clk)
assert(data == 4'b1010);
```

Concurrent Assertion

This assertion is based on clock semantic and use sampled value of their expression. These assertions can expand over multiple cycle.

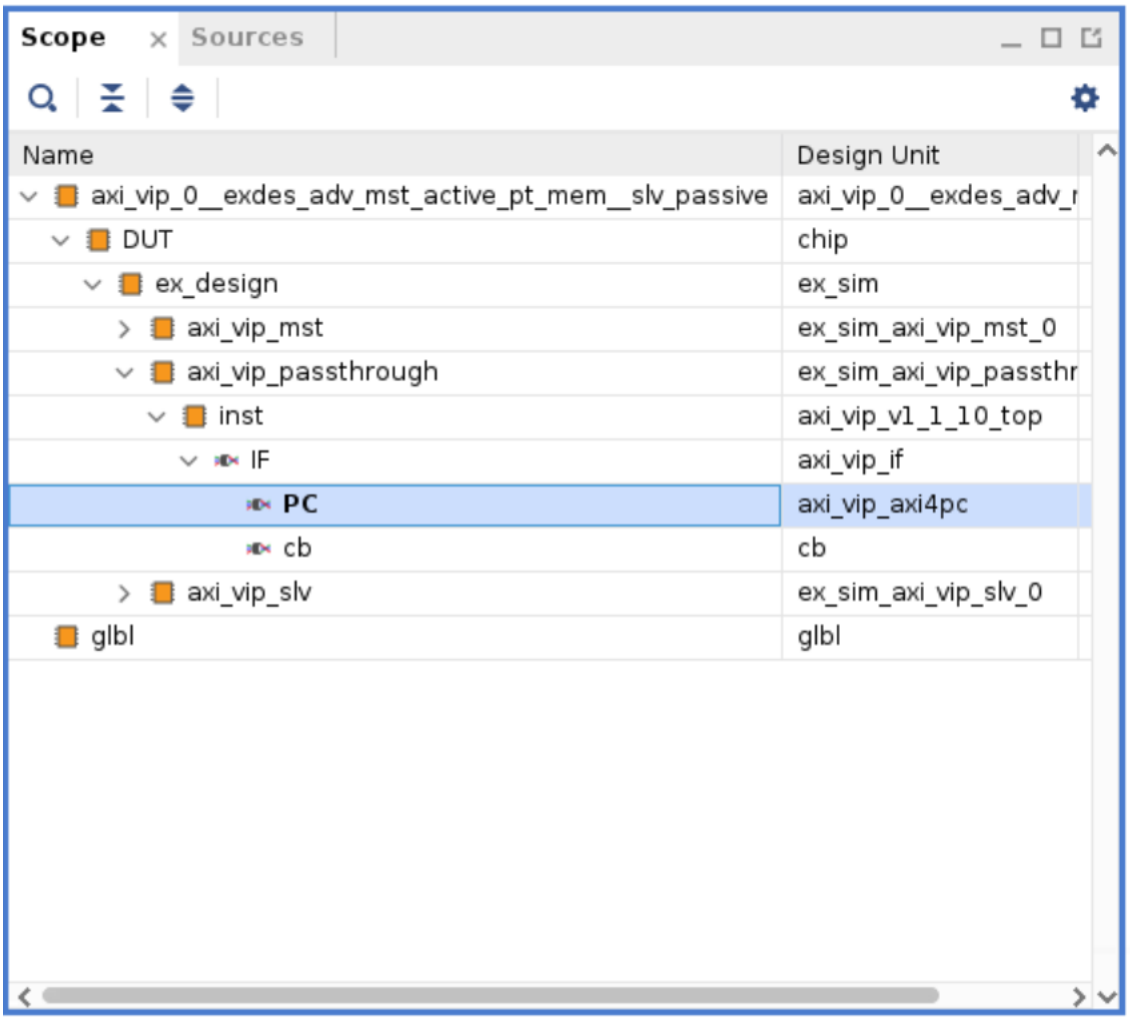
```
always@ (posedge clk)
    a1: assert property (a ##2 b);

a2: assert property (@(posedge clk) a ##2b);
```

In Vivado® simulator, the concurrent assertion of 2nd form that is used outside the procedural block is supported.

1. In Tcl console, invoke `close_sim` command to close the simulation running previously.
2. In Tcl console, invoke `reset_simulation` command to clean the simulation directory.
3. In Tcl console, invoke `launch_simulation` command to run the simulation.
4. In Tcl console, invoke the following command:

```
current_scope /axi_vip_0__exdes_adv_mst_active_pt_mem__slv_passive/DUT/
ex_design/axi_vip_passthrough/inst/IF/PC
```



5. Double-click the scope **PC** to open the source code.
6. Observe that line number 1144 onwards all the property declarations and assertions have been used.

Random Constraint

System Verilog has random constraint, which is used to generate a random value. Using this feature, you can even set the constraint on a random variable.

For each simulation, the simulator is supposed to generate fixed set of values. In this example, randomize call is happening 10 times so each time the simulator is expected to assign a different value on variable 'b1'. If we close the simulation and run it again, the simulator is expected to give same 10 set of values like the previous run. This is called as random stability.

```
module top();
class c1;
rand bit [3:0] b1;
endclass
c1 obj1 = new();
initial
begin
    for(int i = 0; i < 10; i++)
    begin
        #5 obj1.randomize();
        $display("At time %t the value is %p", $time, obj1);
   end
end
endmodule
```

If you want different set of values, you should change the random seed value. In Vivado® simulator, it is done by passing -seed option to xsim. In Tcl Console, you need to invoke the following command:

```
set_property -name {xsim.simulate.xsim.more_options} -value {-seed 2000} -
objects [get_filesets sim_adv_mst_active__pt_passive__slv_comb]
```

With seed, you have to provide any integer value. So just by changing a 'seed' value, you can get a different value. You do not need to do compilation and elaboration again.

1. Add the following code in a file and name it as `random.sv`.

   ```
   module top();
   class c1;
   rand bit [3:0] b1;
   endclass
   c1 obj1 = new();
   initial
   begin
       for(int i = 0; i < 10; i++)
       begin
           #5 obj1.randomize();
           $display("At time %t the value is %p", $time, obj1);
      end
   end
   endmodule
   ```

2. Perform the following in Tcl console:

   a. Invoke `xvlog -sv random.sv` command to compile the code.

   b. Invoke `xelab top -s top` command to elaborate the code.

   c. Invoke `xsim top -R` command to simulate the code.

   Notice the output

run-all

```
At time                  5000 the value is '{b1:3}
At time                 10000 the value is '{b1:7}
At time                 15000 the value is '{b1:7}
At time                 20000 the value is '{b1:0}
At time                 25000 the value is '{b1:0}
At time                 30000 the value is '{b1:5}
At time                 35000 the value is '{b1:9}
At time                 40000 the value is '{b1:3}
At time                 45000 the value is '{b1:12}
At time                 50000 the value is '{b1:0}
exit
```

3. Re-run step 2c and notice the value is similar to the previous one.
4. Simulate the code with different SV seed `xsim top -R -sv_seed 5000` and observe that the value is different. Thus, you can generate different value without going through compile and elaboration step.

Lab 5: Running UVM example

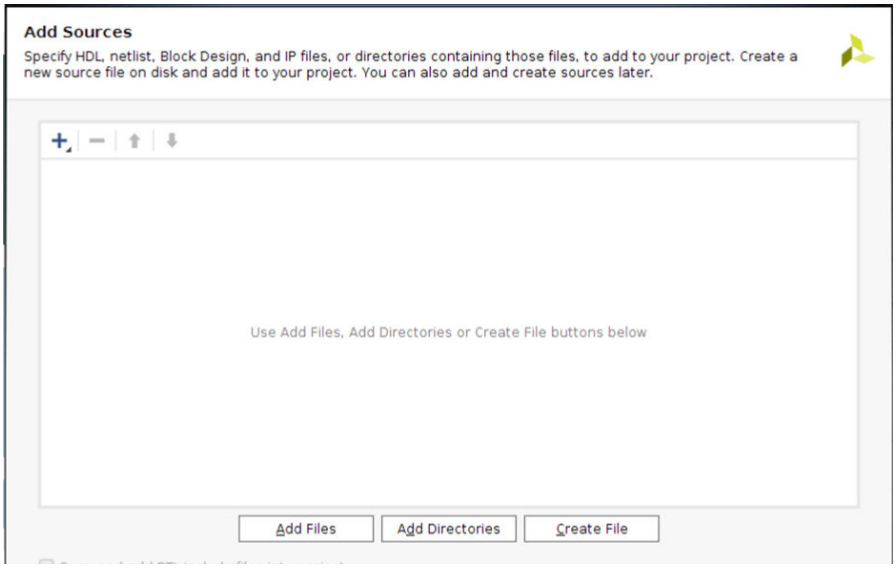
Vivado integrated design environment supports Universal Verification Methodology (UVM) in Vivado simulator (XSIM). UVM version 1.2 is pre-compiled and shipped with Vivado.

Through this tutorial, let's take an UVM based example and run it in Vivado Simulator.

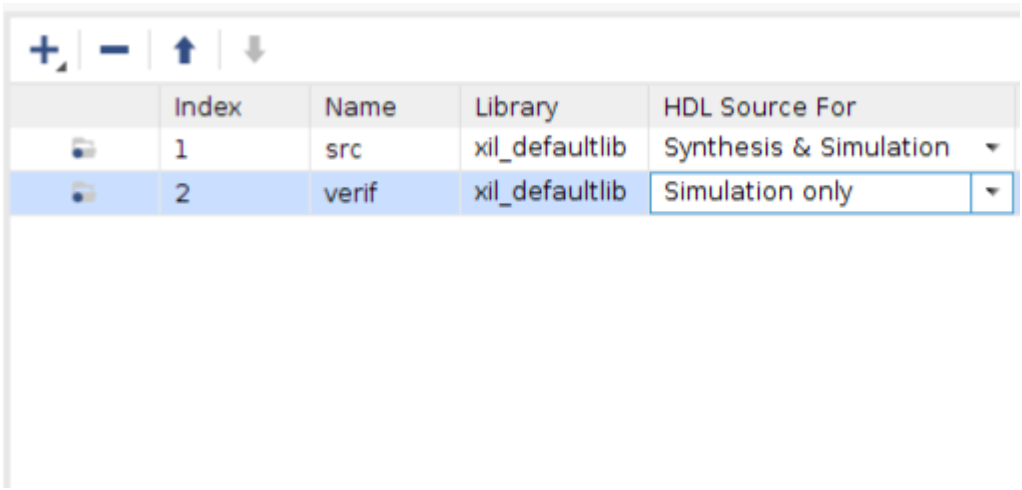
***Note*:** Go to directory `ug937-vivado-design-suite-tutorial-design-files/ug937-design-files/uvm` of tutorial which was downloaded at the start of Chapter 2: Lab 1: Running the Simulator in Vivado IDE.

Creating a New Project

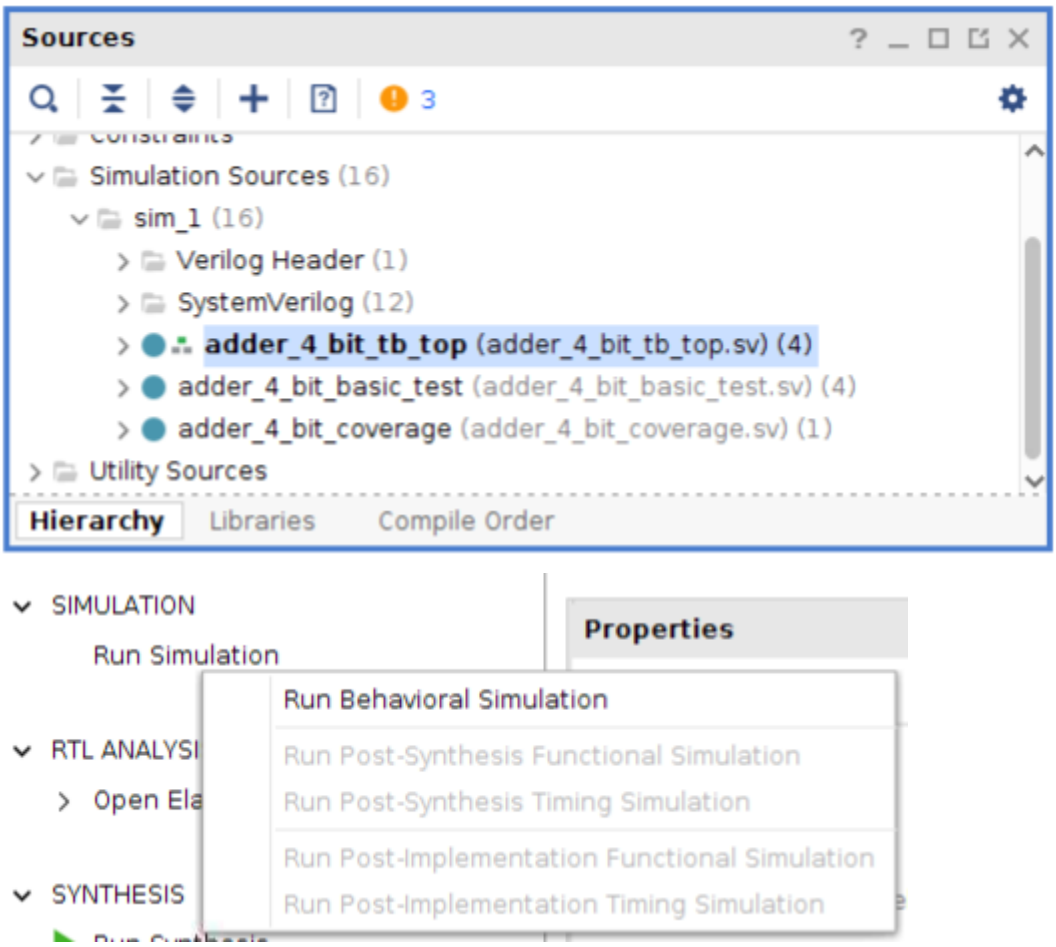
1. Create a new Vivado RTL project in 2021.1, targeting to the required device.
2. Add the directories `src` and `verif` to the project by clicking on the **+** button.



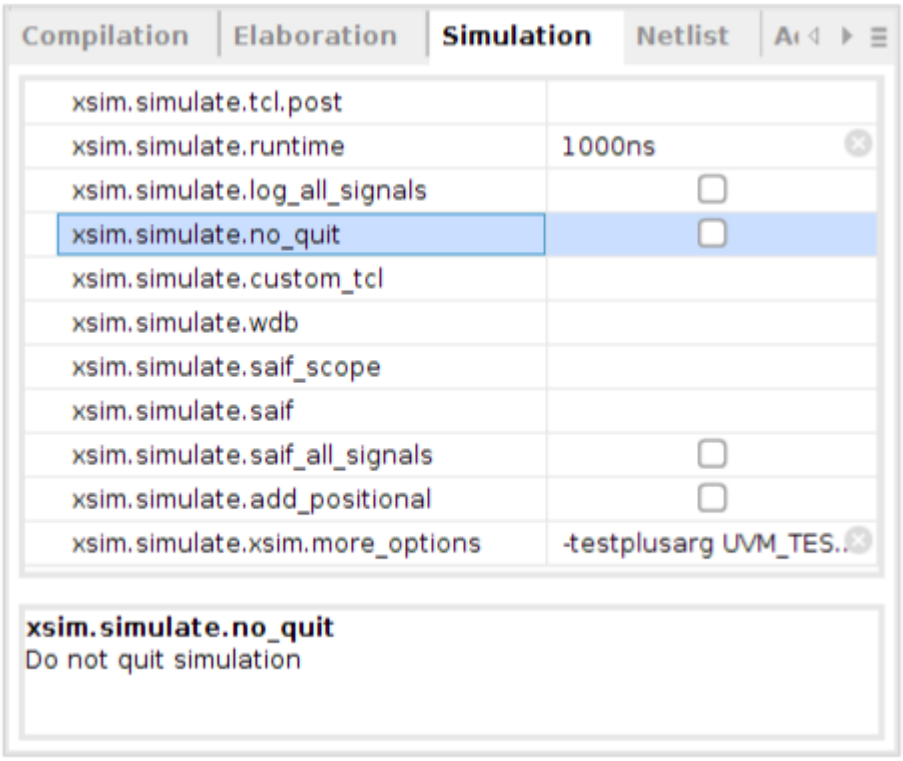
3. Mention the UVM verification files for simulation only.



4. After the hierarchy is updated, ensure to select `adder_4_bit_tb_top.sv` file as top module.



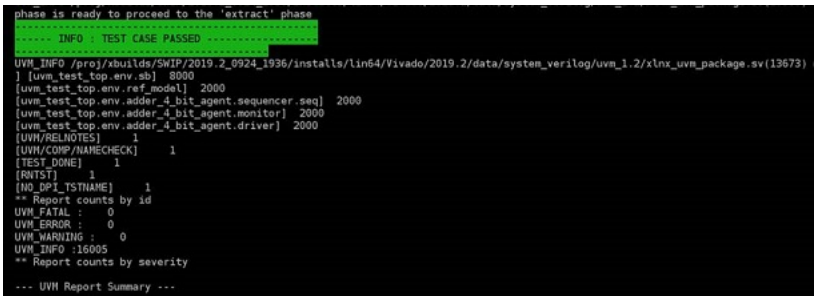
5. For UVM, you need to provide test name, here the test name is `adder_4_bit_basic_test`. For this add `-testplusarg UVM_TESTNAME=adder_4_bit_basic_test -testplusarg UVM_VERBOSITY=UVM_LOW` to `xsim.more_option`



6. Launch simulation. This can run simulation for 1000 ns by default. Click **run all**  .You can see following in TCL console.

Following are the steps to use UVM in Non-Project/Batch Mode:

1. To run the simulation in non-project mode, change the current working directory to the run folder. `cd ./Adder_4_bit/run`
2. For standalone simulation in Vivado you can source `run_xsim.csh` on Linux and `run_xsim.bat` on windows or source `run.tcl` using the below command in Linux/ Windows. Vivado `-mode batch -source run.tcl`
3. Once the simulation gets finished you can observe the UVM test results in the Shell or command prompt as shown in the following figure.



Directory Structure of both project and non project mode:

- src & verif - Design and verification environment related files.
- Run - Location to run simulation in Non project mode.
- UVM_test - Project Mode XSIM simulation.

Lab 6: Running GTM-Wizard example

Overview

PAM4 encodes 2-bits of binary data to into four voltage levels. Through this tutorial, we can create an example to verify designs with PAM4 signals.

Since four voltage levels are not supported by implementation tools. So the PAM4 connections are created as regular single bit wire. For simulation purposes, the PAM4 signals of the design are made accessible in testbench via integer ports of a special module (`xil_dut_bypass`) that gets generated as part of this flow. The generated bypass module is not part of the DUT but has direct access to PAM4 signals of the DUT. This module can be instantiated in testbench to directly drive/observe PAM4 signals.

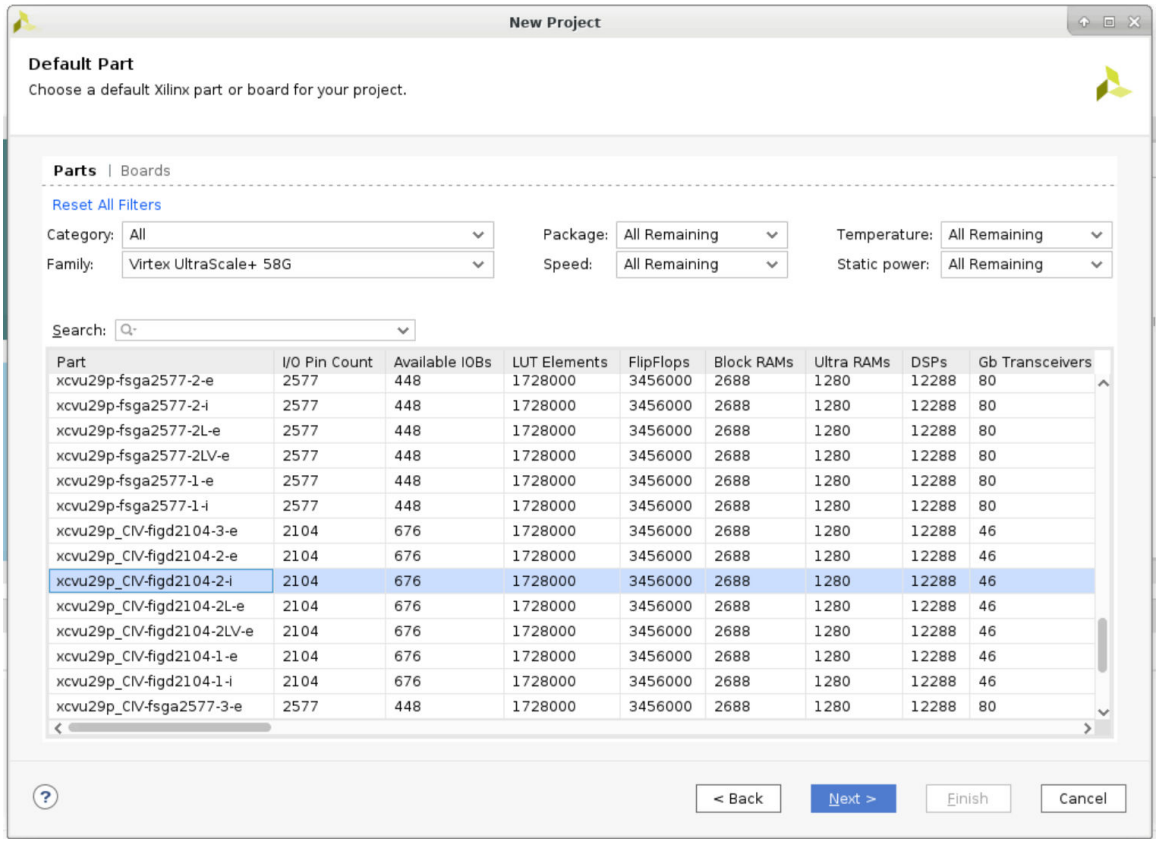
Feature Covered

The following features are covered in the GTM-Wizard:

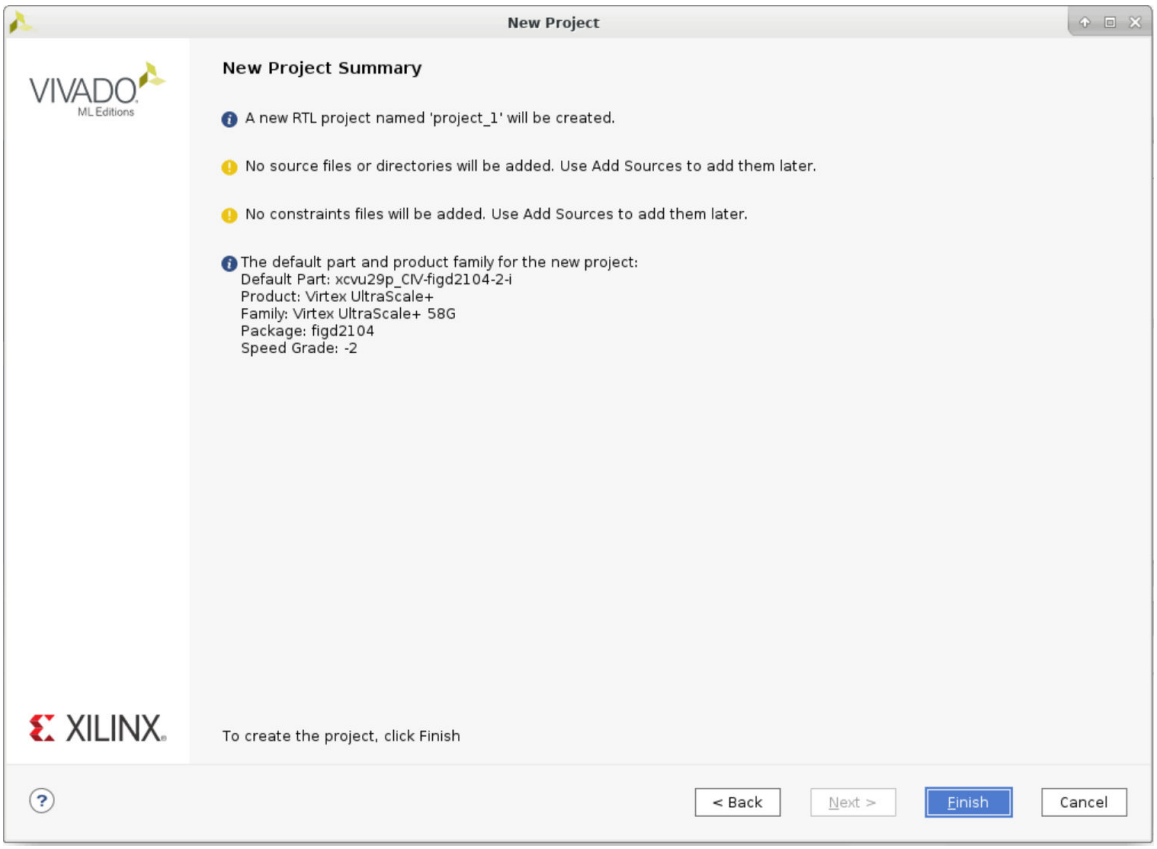
1. Detection of designs with PAM4 signals, designs instantiating GTM_DUAL and automatic generation bypass module (`xil_dut_bypass`).
2. Simple sanity check for design that should instantiate bypass module.
3. A mechanism to view PAM4 signals in Waveform Viewer for XSim users.
4. Provide a way to generate bypass module for export simulation flow.

In this tutorial, we would generate an GTM-Wizard example design, which usage PAM4 signal. To generate that, please follow steps below:

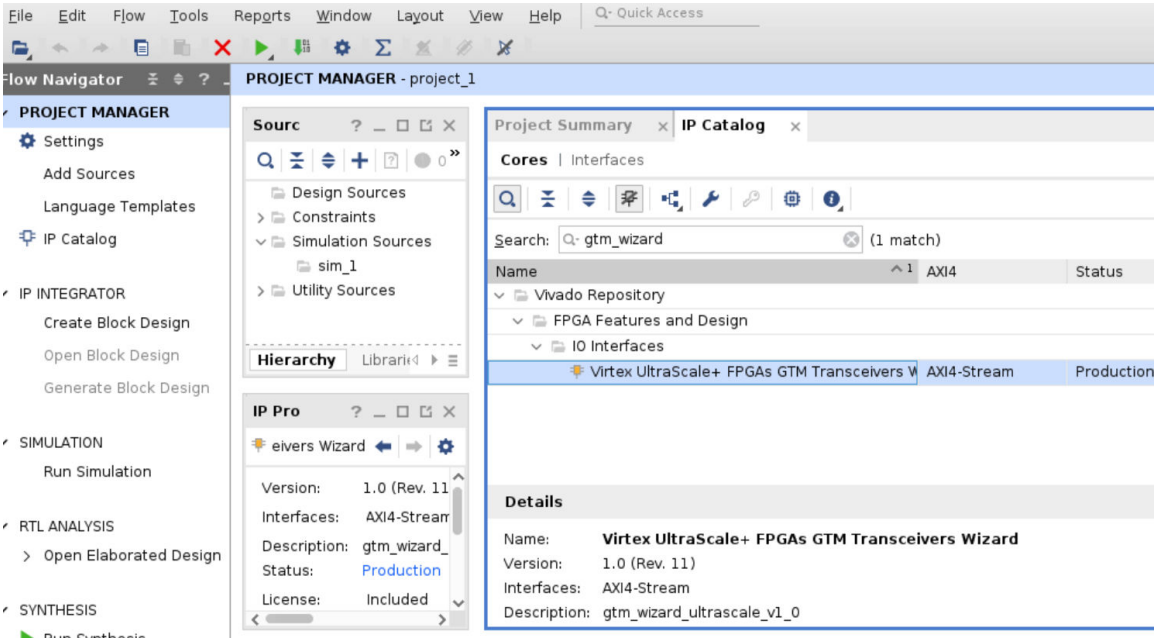
1. Create project in Vivado 2021.1 without adding source/constraint file **Create project → next → next → next → next → next**.
2. In default part, select family Virtex UltraScale+ 58G and select parts as shown in the following figure and click **Next**.



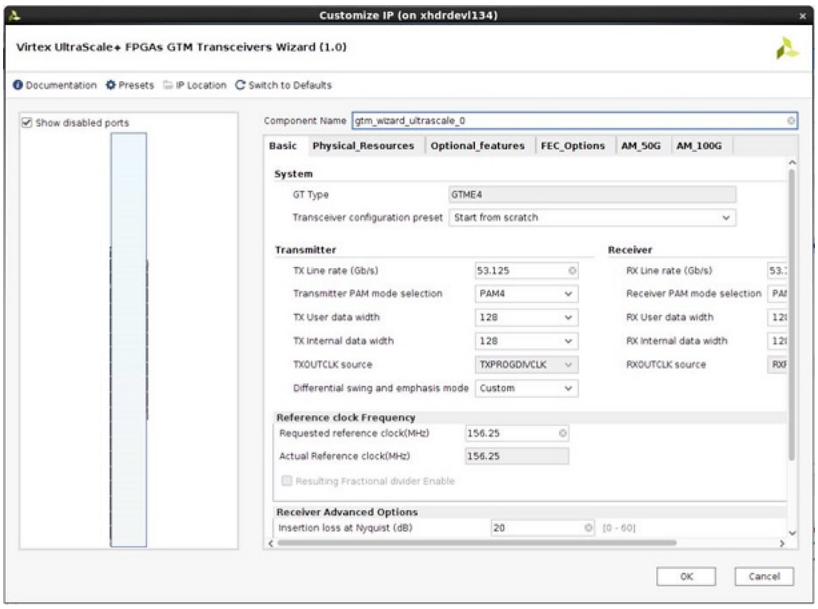
3. Check the summary report and click **finish**.



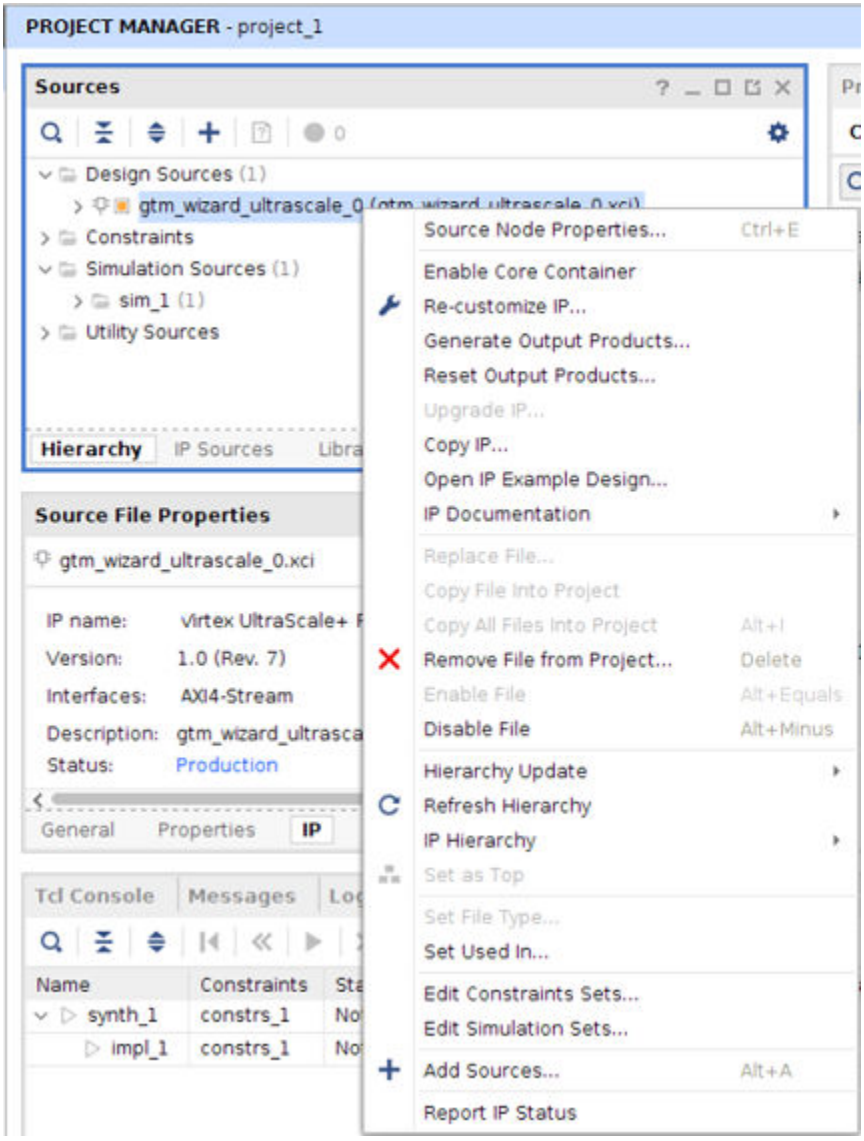
4. Under Project Manager, click on **IP Catalog** and search for `gtm_wizard` and then double click on **block design**.



5. Click **Ok** on default configuration and click **skip** on Generate output product pop-up.



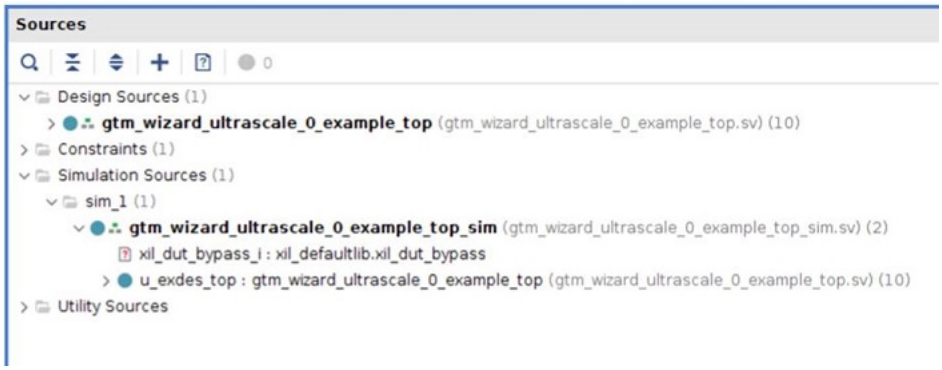
6. On source window, right-click on generated **XCI file** and click open **IP Example Design** by specifying location.



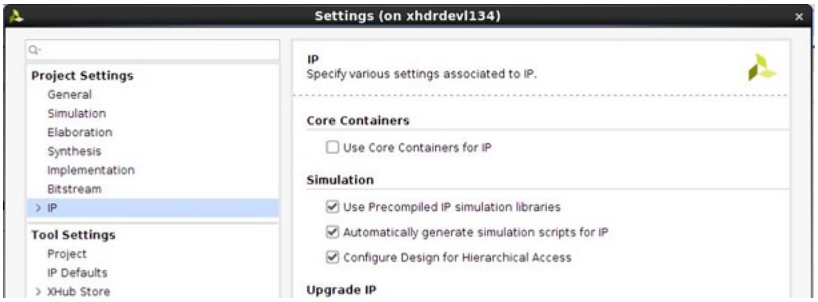
At this stage, you have an example ready to run simulation.

Before heading towards simulation, here are few things from PAM4 point of view:

1. `xil_dut_bypass` module definition is generated on run time by tool which contain hierarchical reference to GTM_DUAL.



2. This `xil_dut_bypass` module generation is control under switch Configure Design for Hierarchical Access, which is set by default.

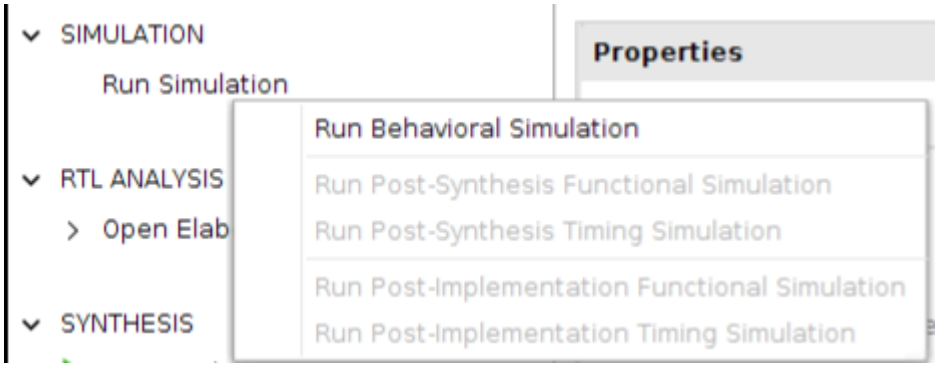


***Note*:** For old behavior, please uncheck Configure Design for Hhierarchical Access.

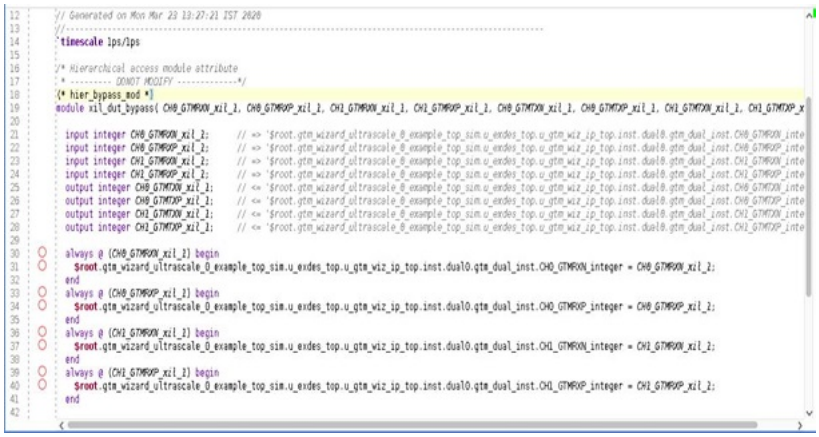
Once design is created, we can run it either through launch_simulation or export simulation.

- **Launch_simulation:**

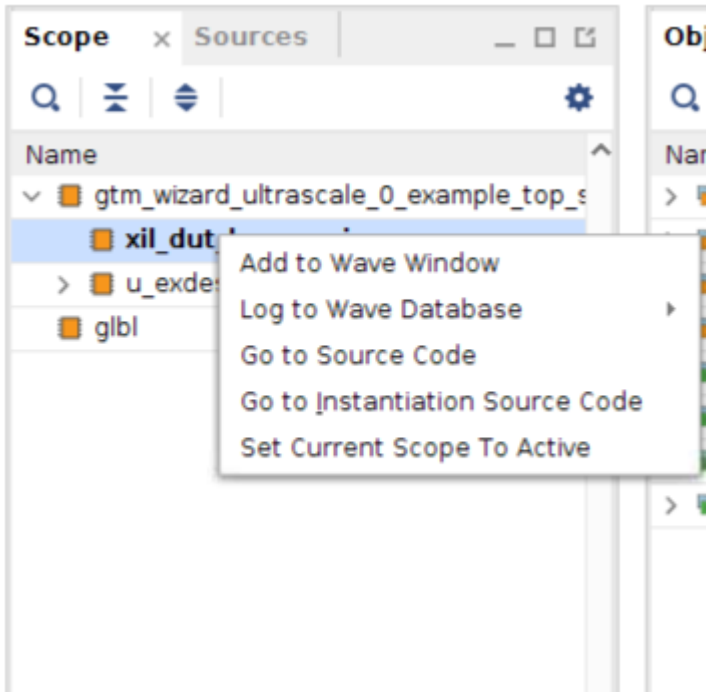
1. Click on **Run Behavioral Simulation**. This will run simulation with Vivado Simulator (XSim).



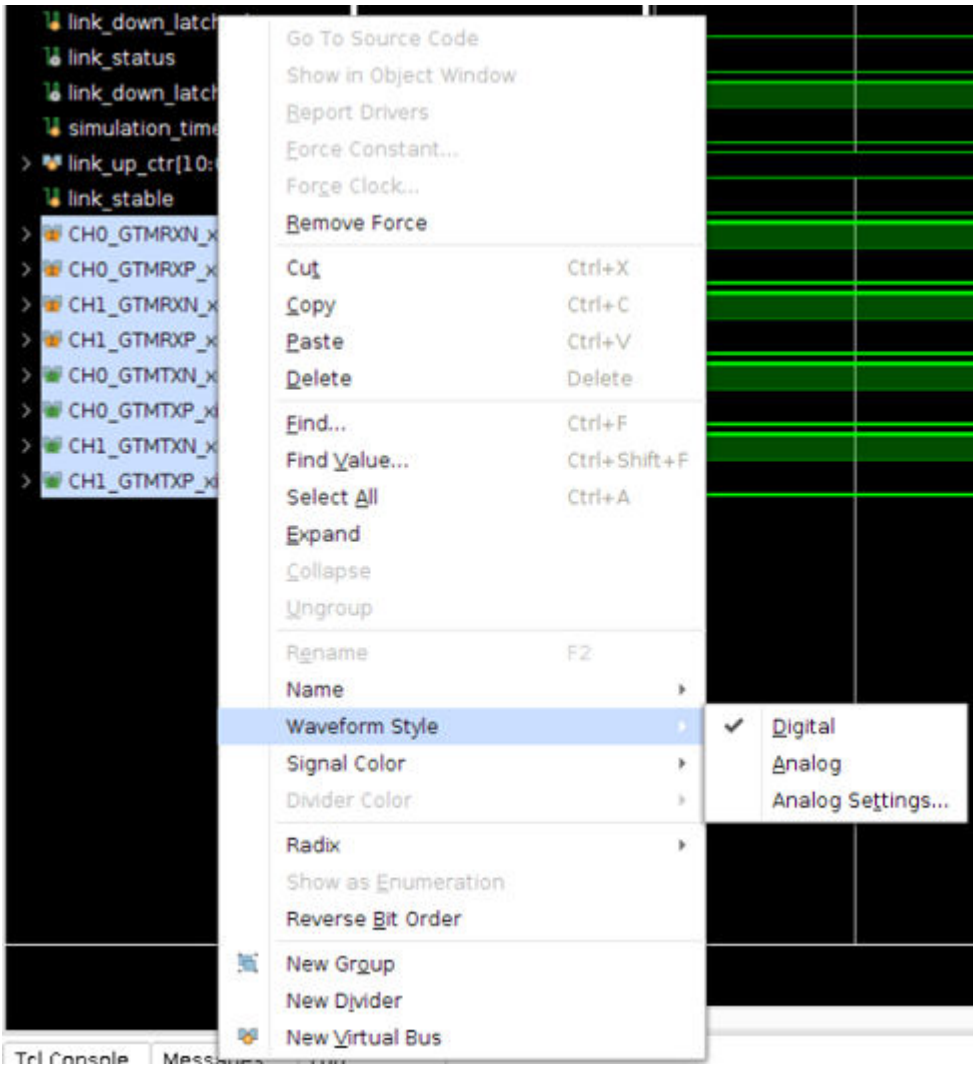
2. Once snapshot is created and loaded, simulation will stop after 1000 ns. Let's look at xil_dut_bypass definition. Double-click on **xil_dut_bypass** in scope window to see the source file. Notice the hierarchical reference from Top module till leaf level instance.



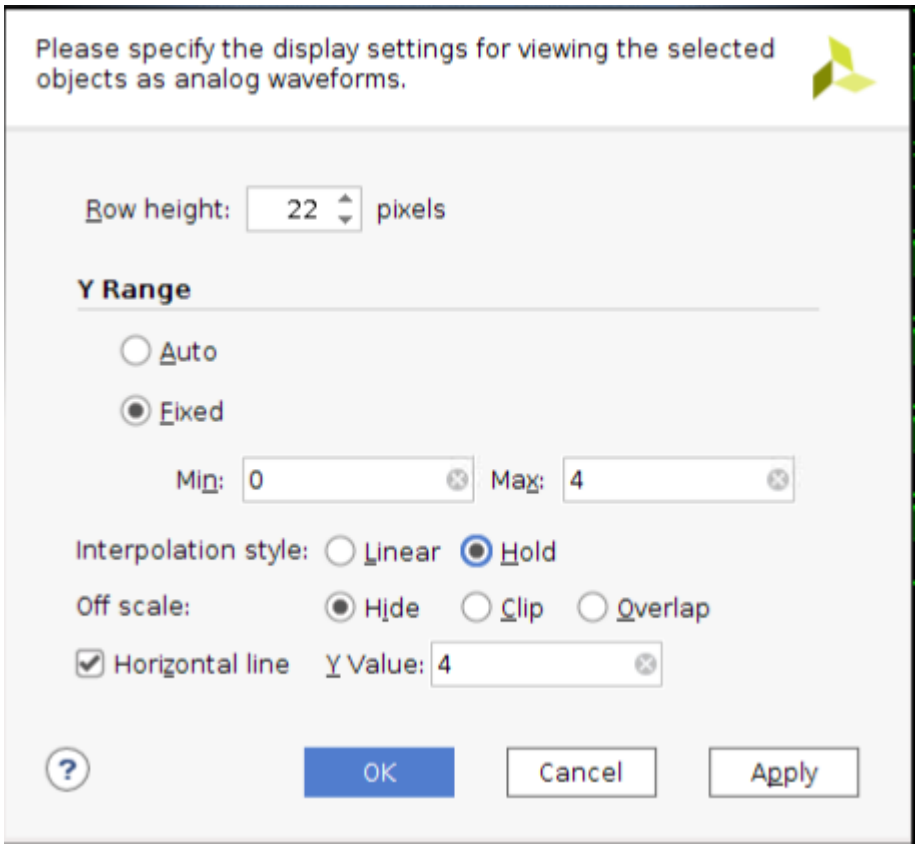
3. Right-click **xil_dut_bypass** and add to waveform.



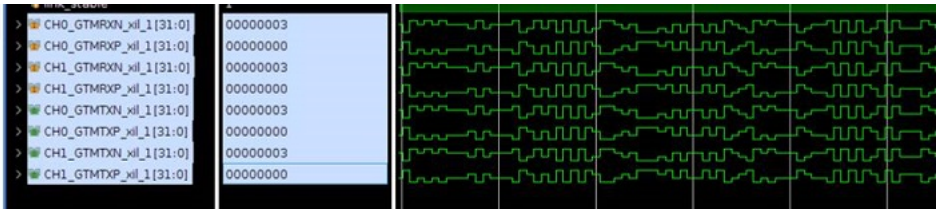
4. Click **restart**  then run all.
5. Once simulation is complete, section all signal of xil_dut_bypass in waveform, right-click and select **waveform style** to **analog**.



6. Go to analog setting under waveform style and change the value like below for better waveform.



7. Observe the analog value of signals.

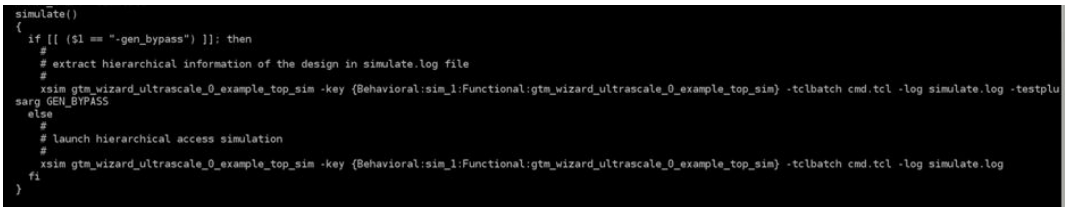


Note: For running same design with third party simulators, please refer to *Vivado Design Suite User Guide: Logic Simulation* (UG900)

- **Export_simulation:** The following are the two flows for export simulation:
  - **Export simulation with xil_dut_bypass generated:**
    1. Invoke command `export_simulation -simulator xsim -generate_hier_access` on Vivado IDE.
    2. This generates scripts in `<your_local_path>/export_sim/xsim`.
    3. In the generated vlog.prj, observe `xil_dut_bypass.sv`. This is because launch_simulation has generated and added this as part of project.

```
sv xil_defaultlib --include "../../hdl" --include "../../gtm_wizard_ultrascale_0_ex.srcs/sources_1/ip/gtm_wizard_ultrascale_0" \
"../../imports/gtm_wizard_ultrascale_0_example_top.sv" \
"../../gtm_wizard_ultrascale_0_ex.srcs/sim_1/imports/hier/xil_dut_bypass.sv" \
"../../imports/gtm_wizard_ultrascale_0_example_top_sim.sv" \
```

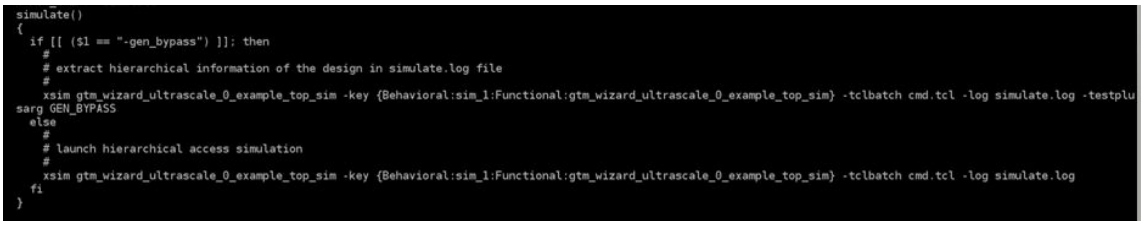
4. Invoke `gtm_wizard_ultrascale_0_example_top_sim.sh` it runs the simulation. If you want to run in GUI mode, add `-gui` to xsim command of else part.



5. Follow step-3 to step-7 under launch_simulation and you will able to see same waveform/output as launch_simulation.

- **Export simulation without xil_dut_bypass:**

1. Invoke `gtm_wizard_ultrascale_0_example_top_sim.sh -gen_bypass`.
2. It runs the simulation for delta time unit and generate hierarchical path in log file.
3. In generated simulate.log, note down the entry `xilinx_hier_bypass_ports:gtm_wizard_ultrascale_0_example_top_sim.u_e….`
4. Generate `xil_dut_bypass.sv` by invoking `generate_hier_access -log ./simulate.log` on Vivado TCL console.
5. Observe `xil_dut_bypass.sv` generate in current directory.
6. Add this `xil_dut_bypass.sv` in vlog.prj as `sv xil_defaultlib ./xil_dut_bypass.sv`.
7. Invoke `gtm_wizard_ultrascale_0_example_top_sim.sh` it runs simulation. If you want to run in GUI mode, add `-gui` to xsim command of else part.



8. Follow step-3 to step-7 under launch_simulation and you can see same waveform/output as launch_simulation.

- **Generating xil_dut_bypass for non-vivado project:**

1. Create compile order of the design (<design>.prj).
2. Execute XSim simulator tools to generate the simulator log file:
   a. `xelab -prj <design>.prj -top <testbench-top>`.
   b. `xsim -R <testbench-top> --testplusarg GEN_BYPASS`.

3. Verify `<simulator>.log` file generated and that it contains the `xilinx_hier_bypass_ports` string with the hierarchical path information. For example:

```
xilinx_hier_bypass_ports:tb.dut_i.gtmWiz_00.gtm_i
in:integer:in1:in_var1 in:integer:in2:in_var2
out:integer:out1:out_var1 out:integer:out2:out_var2
```

4. Download `generate_hier_access.tcl` utility from GitHub:

```
wget https://raw.githubusercontent.com/Xilinx/XilinxTclStore/2020.1-
dev/tclapp/xilinx/projutils/generate_hier_access.tcl
```

5. Execute `generate_hier_access.tcl` to generate the sources for hierarchical access simulation:
   a. `# /usr/bin/tclsh`.
   b. `source generate_hier_access.tcl`.
   c. `generate_hier_access -bypass dut_bypass -testbench <module> -directory <path> -log <simulator>.log`. Instantiate this `xil_dut_bypass` in test bench with proper connection.
6. Add `<path>/xil_dut_bypass.sv` to <design>.prj.
7. Run simulator tools to simulate the design in <design>.prj.

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

Please Read: Important Legal Notices

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 Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; 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 such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright