Vivado Design Suite User Guide:

Synthesis

UG901 (v2012.2) July 25, 2012

XILINX®

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

© Copyright 2012 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
07/25/12	2012.2	Initial Xilinx Release of the Vivado™ User Guide: Synthesis.

Table of Contents

Revision History 2

Vivado Synthesis

Introduction 4
Synthesis Methodology 5
Using Project Mode 5
Viewing Floorplanning and Reporting Resource Statistics 21
Exploring the Logic 25
About XST Strategies 28
Using Non-Project Mode for Synthesis 29

Appendix A: Synthesis Attributes

Introduction 31
Supported Attributes 31

Appendix B: SystemVerilog Support

Introduction 40
Targeting SystemVerilog for a Specific File 40
Data Types 40
Processes 45
Procedural Programming Assignments 47
Tasks and Functions 49
Modules and Hierarchy 50
Interfaces 51

Appendix C: Additional Resources

Xilinx Resources 55
Solution Centers 55
Vivado Documentation 55

Vivado Synthesis

Introduction

Synthesis is the process of transforming an RTL-specified design into a gate-level representation. Vivado™ Integrated Design Environment (IDE) synthesis is timing-driven and optimized for memory usage and performance. Support for SystemVerilog as well as mixed VHDL and Verilog languages is included. The tool supports Xilinx® Design Constraints (XDC), which is based on the industry-standard Synopsys Design Constraints (SDC).

IMPORTANT: *UCF constraints are not supported with Vivado synthesis. Migrate UCF constraints to XDC constraints. For more information, see the "UCF to XDC Constraints Conversion" in the Vivado Design Suite Migration Methodology Guide (UG912) [Ref 4].*

There are two ways to setup and run synthesis:

- Use *Project Mode*, from the Vivado IDE
- Use *Non-Project Mode*, applying the `synth_design` Tool Command Language (Tcl) command and controlling your own design files.

See the *Vivado Design Suite User Guide: Design Flows Overview (UG892)* [Ref 8] for more information about operation modes. This chapter covers both modes in separate subsections.

Synthesis Methodology

The Vivado IDE includes a synthesis and implementation environment that facilitates a push-button flow with synthesis and implementation runs. The tool manages the run data automatically, allowing repeated run attempts with varying Register Transfer Level (RTL) source versions, target devices, synthesis or implementation options, and physical or timing constraints. Within the Vivado IDE, you can:

- Create and save *strategies*. Strategies are configurations of command options, that you can apply to design runs for synthesis or implementation. See Creating Run Strategies, page 8.
- Queue the synthesis and implementation runs to launch sequentially or simultaneously with multi-processor machines. See Running Synthesis, page 14.
- Monitor synthesis or implementation progress, view log reports, and cancel runs. See Monitoring the Synthesis Run, page 18.

Using Project Mode

This section describes using the Vivado IDE to set up and run Vivado Synthesis.

Using Synthesis Settings

To set the synthesis options for the design, from the Synthesis section of the Flow Navigator:

1. Click the **Synthesis Settings** button, as shown in Figure 1.

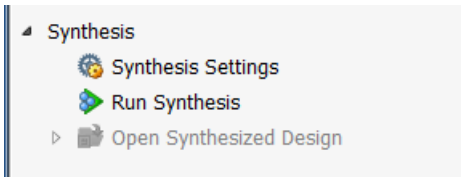

Figure 1: **Flow Navigator: Synthesis**

The Project Settings dialog box opens, as shown in Figure 2, page 6.

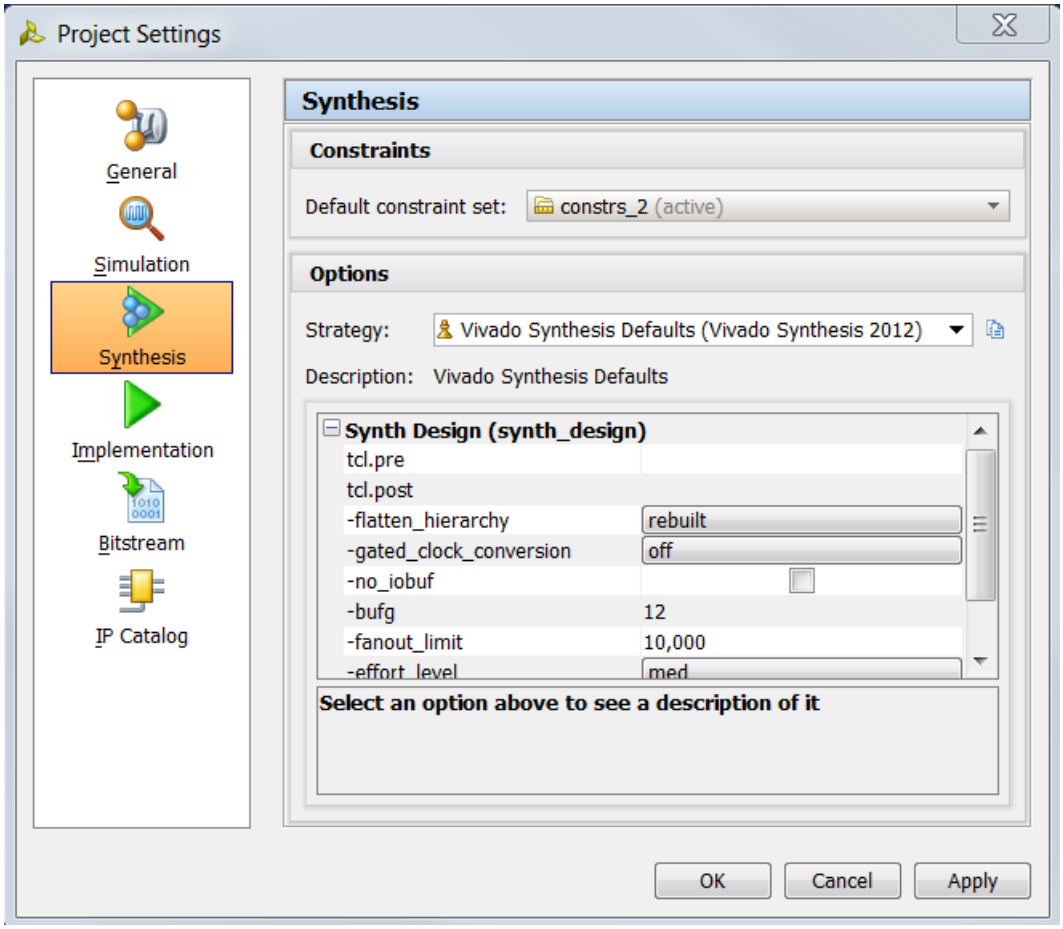

Figure 2: **Project Settings Dialog Box**

2. From the Project Setting dialog box, select:

   a. From Synthesis **Constraints**: Select the **Default Constraint Set** as the active *constraint set*. A constraint set is a set of files containing design constraints captured in Xilinx Design Constraints (XDC) files that can be applied to your design. The two types of design constraints are:

      - Physical constraints define pin placement, and absolute, or relative, placement of cells such as block RAMs, LUTs, Flip-Flops, and device configuration settings.
      - Timing constraints, written in industry standard SDC, define the frequency requirements for the design. Without timing constraints, the Vivado Design Suite optimizes the design solely for wire length and placement congestion.

      The selected constraint set is used for new runs, and it is also the constraint set that is targeted for design changes.

   b. From the **Options** area: Select a **Strategy** from the drop-down where you can view and select a predefined synthesis strategy to use for the synthesis run.

You can also define your own strategy. When you select a synthesis strategy, the command-line options for Vivado or XST display in the lower part of the dialog box. You can override synthesis strategy settings by changing the option values.

Figure 3 show the Strategy drop-down with the Vivado Synthesis option highlighted.

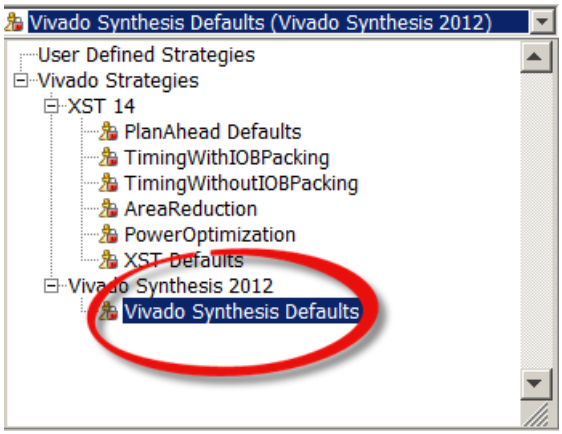

Figure 3: **Vivado Synthesis Defaults Drop-Down Box**

Xilinx recommends the Vivado Synthesis 2012 strategies. There is more information about XST strategies in About XST Strategies, page 28.

c. Select from the displayed **Options**, which are:

The **tcl.pre** and **tcl.post**, options are hooks for Tcl files that run immediately before and after synthesis.

-flatten_hierarchy: Determines how Synthesis controls hierarchy.

`none`: Instructs the synthesis tool to never flatten the hierarchy. The output of synthesis will have the exact same hierarchy as the original RTL.

`full`: Instructs the tool to fully flatten the hierarchy leaving only the top level.

`rebuilt`: Is the default option. When set, rebuilt allows the synthesis tool to flatten the hierarchy, perform synthesis, and then rebuild the hierarchy based on the original RTL. This value allows the QoR benefit of cross-boundary optimizations, with a final hierarchy that is similar to the RTL for ease of analysis.

-gated_clock_conversion: Turns on and off the Synthesis tools ability to convert `clock_logic` with enables. The use of gated clock conversion also requires the use of an RTL attribute to work. See Appendix B, SystemVerilog Support, for more information.

–no_iobuf: Instructs the tool to not infer any input or output buffers. This is useful in a bottom-up flow where the output of the tool is used as a lower level later in the flow. This setting is a global setting that includes the whole design.

To use this on a port-by-port basis, use the `BUFFER_TYPE` attribute described in the Appendix A, Synthesis Attributes.

–bufg: Controls how many `BUFGs` the tool infers in the design. This option is used when other `BUFGs` in netlists are not visible to the synthesis process.

The tool infers up to the amount specified, and tracks how many `BUFGs` are instantiated in the RTL. For example, if the `–bufg` option is set to 12 and there are 3 `BUFGs` instantiated in the RTL, the tool infers up to 9 more `BUFGs`.

–fanout_limit: Specifies the number of loads a signal must drive before it starts replicating logic. This global limit is a general guide, and when the tool determines it is necessary, it can ignore the option. If a hard limit is required, see the `MAX_FANOUT` option described in the Appendix A, Synthesis Attributes.

–fsm_extraction: Controls how synthesis extracts and maps finite state machines. By default this option is set to `off`, and the state machine is synthesized as logic. Or you can choose from the following options to encode the state machine in a specific encoding type: `one_hot`, `sequential`, `johnson`, `gray`, or `auto`.

Creating Run Strategies

A strategy is a set of switches to the tools, which are defined in a pre-configured set of options for the synthesis application or the various utilities and programs that run during implementation. Strategies are tool- and version-specific. Each major release has version-specific options.

To see the current strategies for the flow, select **Tools > Options** and click the **Strategies** button on the left to open the strategies window, as shown in Figure 4, page 9.

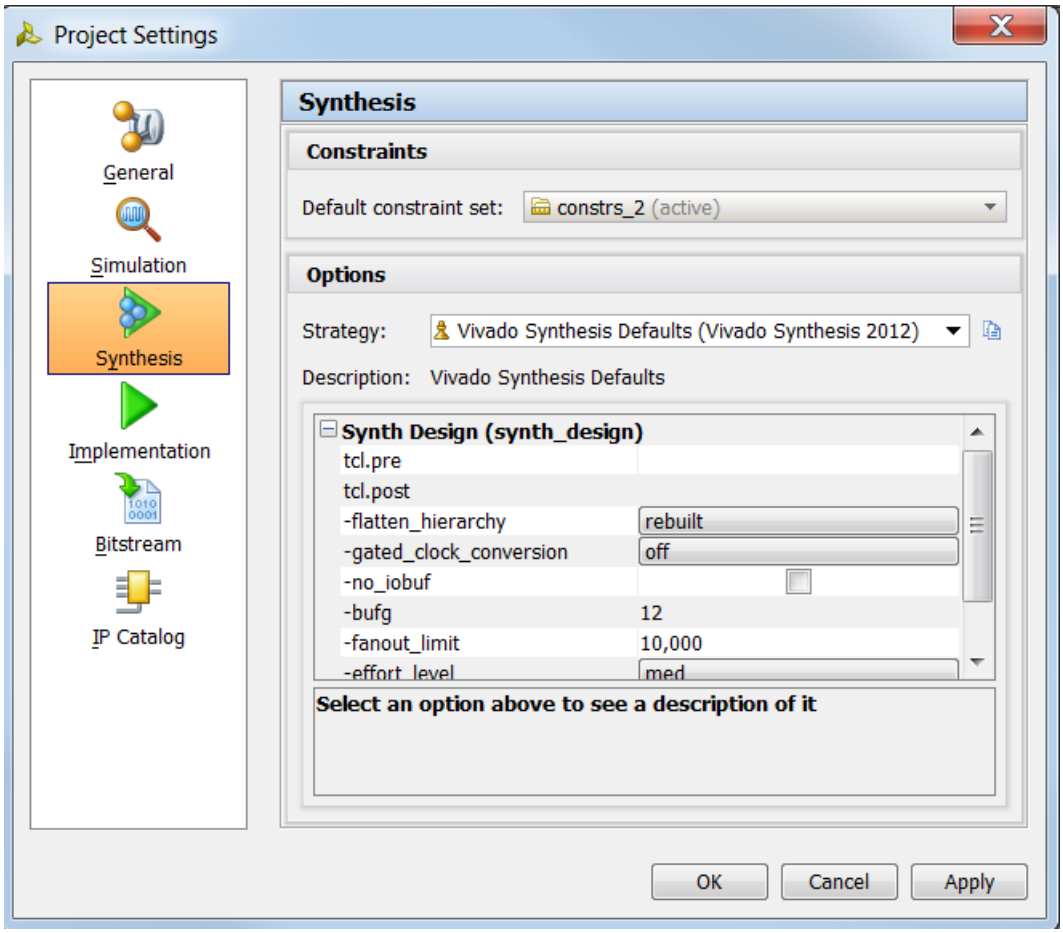

Figure 4: **Synthesis Strategies**

In the **Flow** drop-down, select **Vivado Synthesis**. The options on the right are the same as those shown in the Synthesis Project Settings dialog box.

To create a custom strategy, do one of the following:

- Right-click the **User Defined Strategies** > **Create New Strategy**
- Click the **Create New Strategy** button under the flow options to open the New Strategy dialog box, as shown in Figure 5.

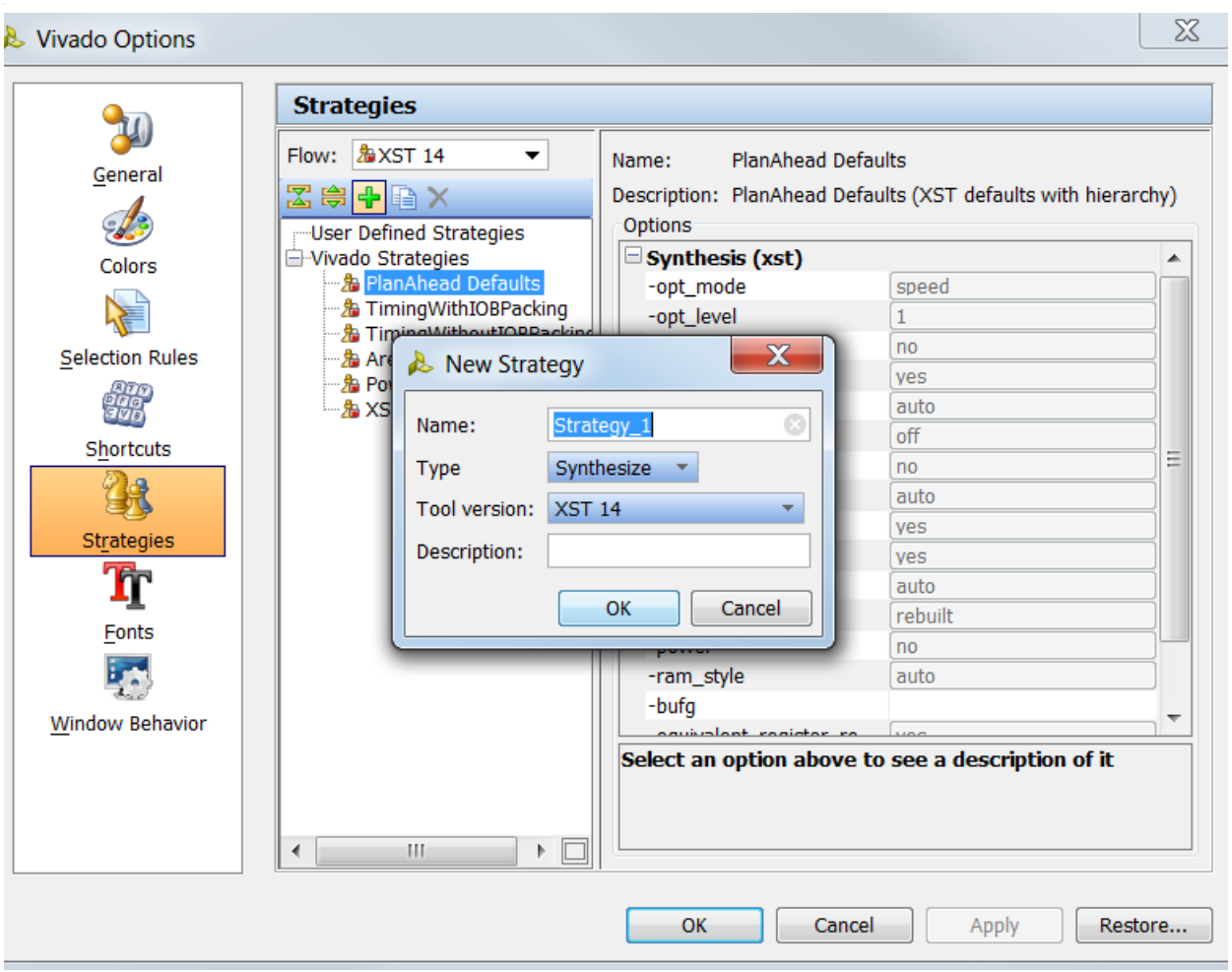

Figure 5: **New Strategy Dialog Box**

From the New Strategy dialog box, name your strategy, set the strategy type, and specify the tool version. There is an input option for a meaningful description. After you set the options, click **OK**.

Setting Synthesis Inputs

Vivado synthesis allows two input types: RTL source code and timing constraints.

To add RTL or constraint files to the run, in the **Synthesis** section of the Flow Navigator, select the **Add Sources** command to open the Add Sources wizard, shown in Figure 6.

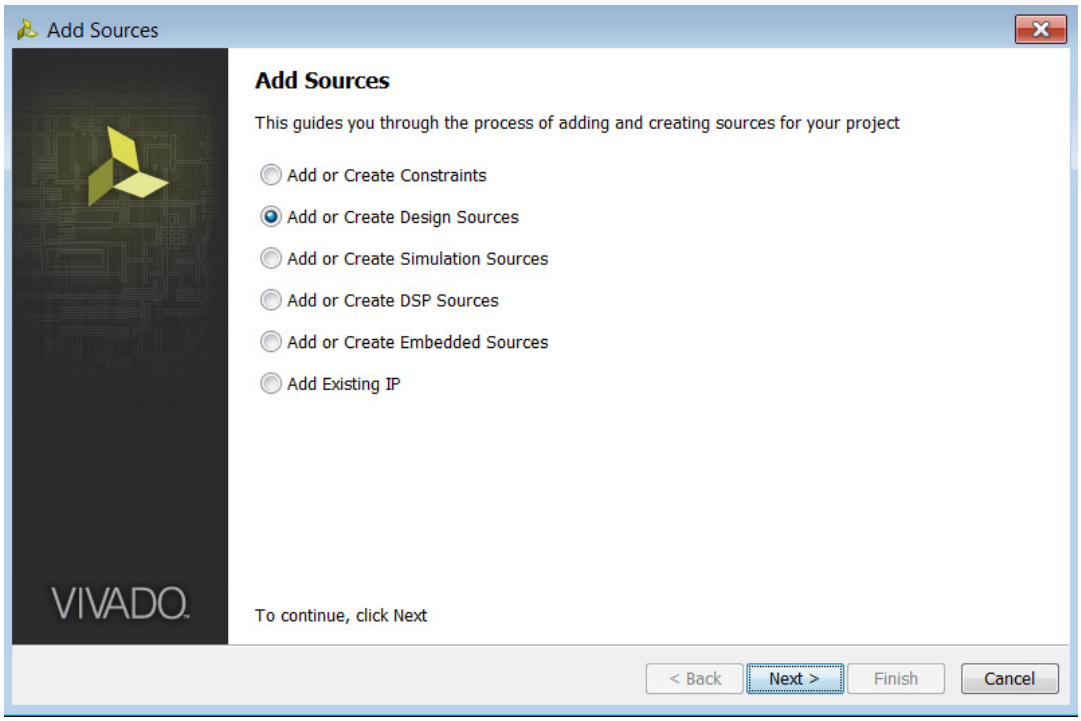

Figure 6: **Add Sources Wizard**

Add constraint, RTL, or other project files. See the *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)* [Ref 5] for more information about the Add Source wizard.

The Vivado Synthesis tool can read the synthesizable subset of files in VHDL, Verilog, or SystemVerilog, which is supported in the Xilinx tools. Appendix B, SystemVerilog Support, provides details on which SystemVerilog constructs are supported.

Vivado Synthesis also supports several RTL attributes that control synthesis behavior. These attributes are described in Appendix A, Synthesis Attributes.

Vivado synthesis uses the XDC file for timing constraints.

IMPORTANT: *Vivado Design Suite does not support the UCF format. See the Vivado Design Suite Migration Methodology Guide (UG912) [Ref 7] for the UCF to XDC conversion procedure.*

Controlling File Compilation Order

A specific compile order is necessary when one file has a declaration and another file depends upon that declaration. The Vivado IDE controls RTL source files compilation from the top of the graphical hierarchy shown in the Sources window Compile Order window to the bottom.

The Vivado IDE automatically identifies and sets the best top-module candidate. The compile order is also automatically managed. The top-module file and all sources that are under the active hierarchy are passed to synthesis and simulation in the correct order.

In the Sources window, a popup menu provides the **Hierarchy Update** command. The options provided specify to the Vivado IDE how to handle changes to the top module and to the source files in the design.

The default setting, **Automatic Update and Compile Order**, specifies that the tool:

- Manages the compilation order as shown in the Compilation window
- Shows which modules are used and where they are in the hierarchy tree in the Hierarchy window

The compilation order automatically updates as you change source files.

To modify the compile order before synthesis:

1. Set **Hierarchy Update > Automatic Update, Manual Compile Order so** the Vivado IDE automatically determines the best top module for the design and allows manual specification of the compilation order.
2. Drag and drop files in the **Compile Order** window of the Sources window popup menu to arrange the compilation order, or use the **Move Up** or **Move Down** commands in the Sources window popup menu.

   See the *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)* [Ref 3] for information about the Sources window.

Defining Global Include Files

The Vivado IDE supports designating one of more Verilog or Verilog Header source files as global `include` files. Files that are marked as global `` `include `` are processed before any other sources.

Verilog typically requires an `` `include `` statement to be placed at the top of any Verilog source file that references content from a another Verilog or header file. Designs that use common header files might require multiple `` `include `` statements to be repeated across multiple Verilog sources used in the design.

To designate a Verilog or Verilog header file as a global `` `include `` file:

1. In the Sources window, select the file and open the popup menu.
2. Select the **Set Global Include** command, or use the **Global Include** checkbox in the Source File Properties window, as shown in Figure 7.

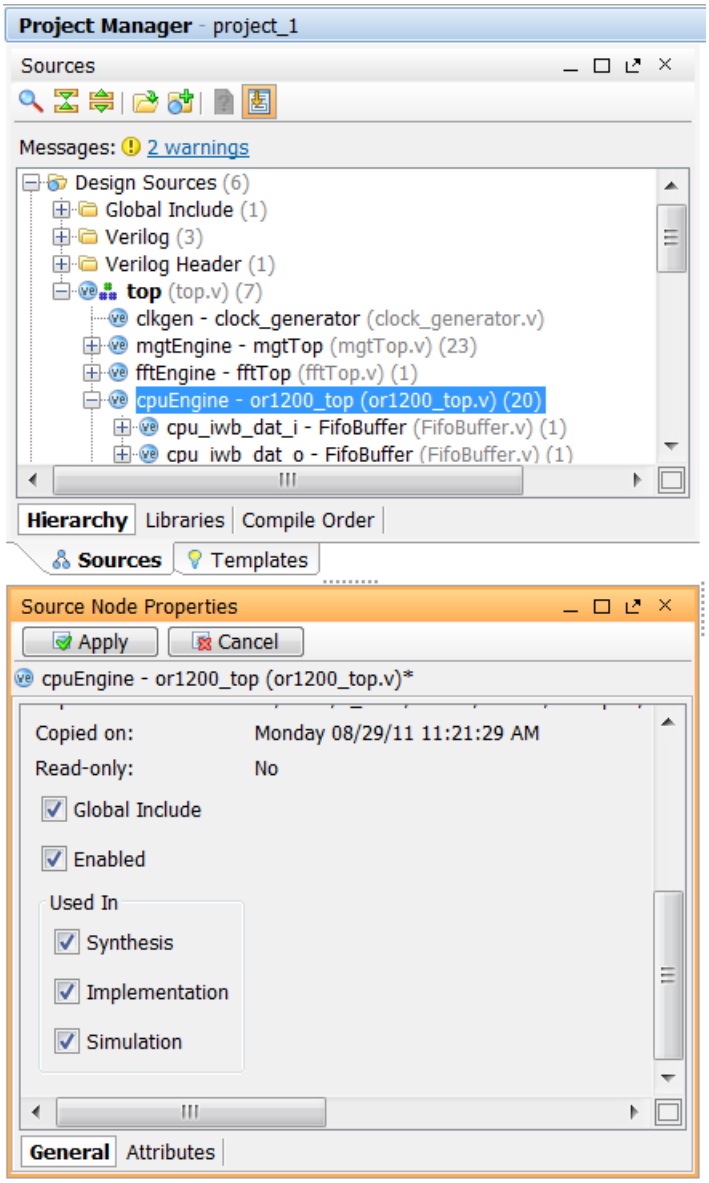

Figure 7: **Source Node Properties Window**

TIP: *Reference header files in Verilog that should be specifically applied to a single Verilog source (for example; a particular `define macro), with an `include statement instead of marking it as a global include file.*

See the *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)* [Ref 3], for information about the Sources window.

Running Synthesis

A *run* defines and configures aspects of the design that are used during synthesis. A synthesis run defines the:

- Xilinx device to target during synthesis
- Constraint set to apply
- Options to launch single or multiple synthesis runs
- Options to control the results of the synthesis engine

To define a run of the RTL source files and the constraints:

1. From the main menu, select one of the following:
   - **Flow > Create Runs**
   - From the Flow Navigator, right-click Run Synthesis and select **Create Synthesis Runs** from the popup menu.

   The Create New Runs wizard opens. The first page of the wizard is a summary of the command.

2. Click **Next**.

   The Create New Runs dialog box opens, as shown in Figure 8.

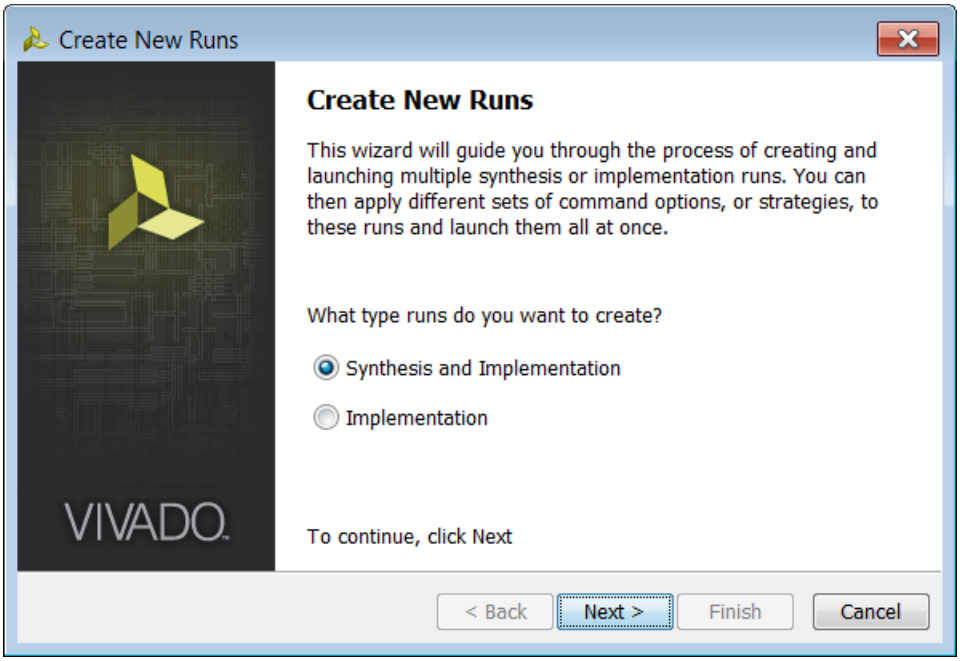

Figure 8: **Create New Runs Dialog Box**

3. Select the **Synthesis and Implementation** checkbox and click **Next.**

The Set-Up Synthesis Run dialog box opens, as shown in Figure 9.

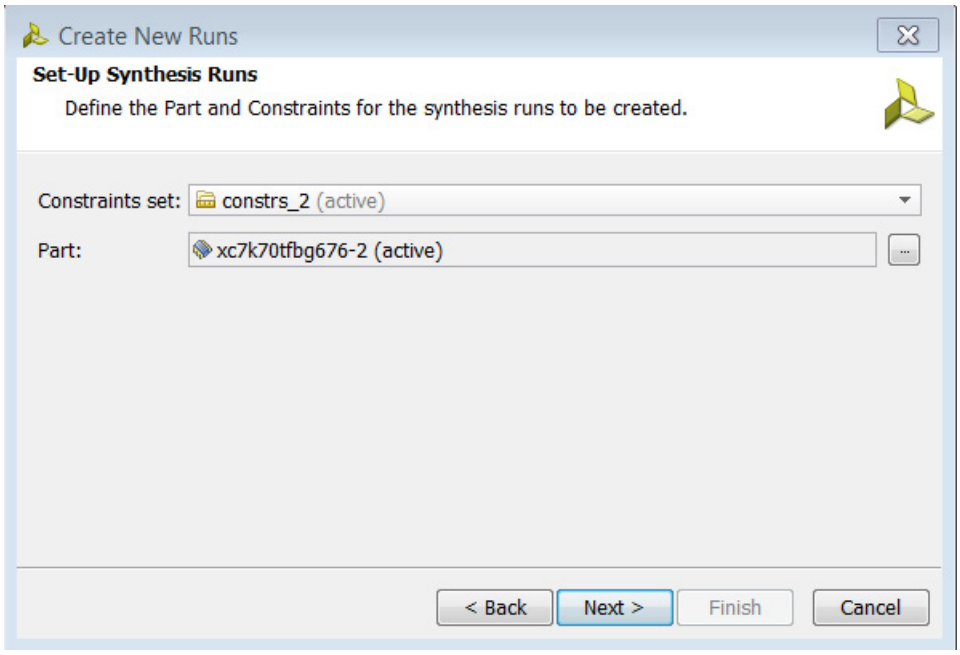

Figure 9: **Set-Up Synthesis Runs Dialog Box**

4. Select the **Constraints set** and **Part**, then click **Next**.

   The Choose Synthesis Strategies dialog box opens, as shown in Figure 10.

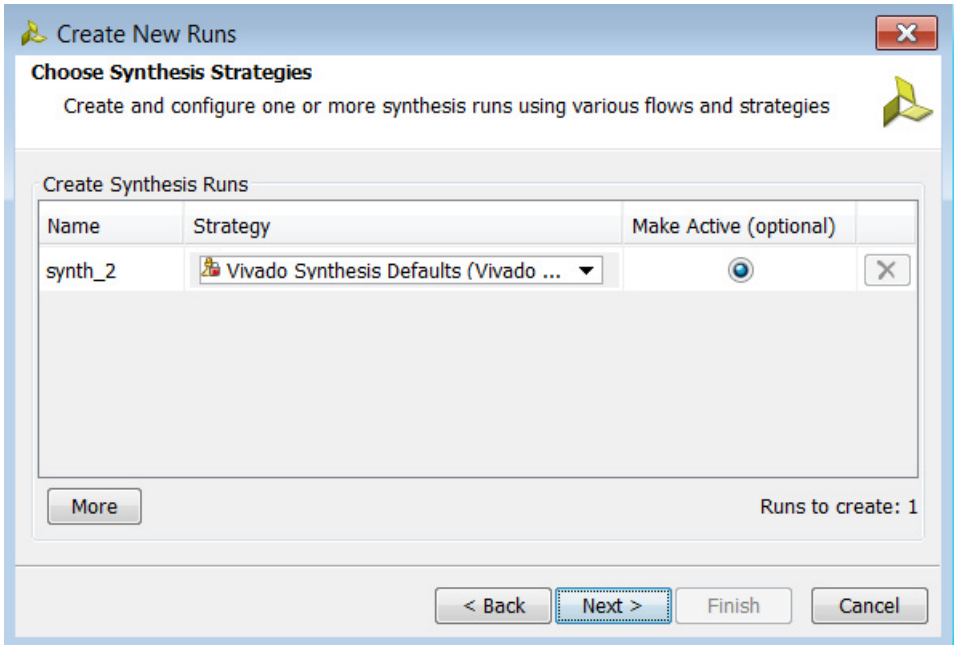

Figure 10: **Choose Synthesis Strategies Dialog Box**

5. Select a defined strategy.

The Vivado IDE contains a default strategy. You can set a specific name for the strategy run or accept the default name(s), which are `synth_1`, `synth_2`, and so forth.

TIP: Xilinx Synthesis Technology (XST) is available as a strategy also. See About XST Strategies, page 28 for information about XST strategies and reports.

To create your own run strategy, see Creating Run Strategies, page 8.

The Launch Options dialog box opens, as shown in Figure 11.

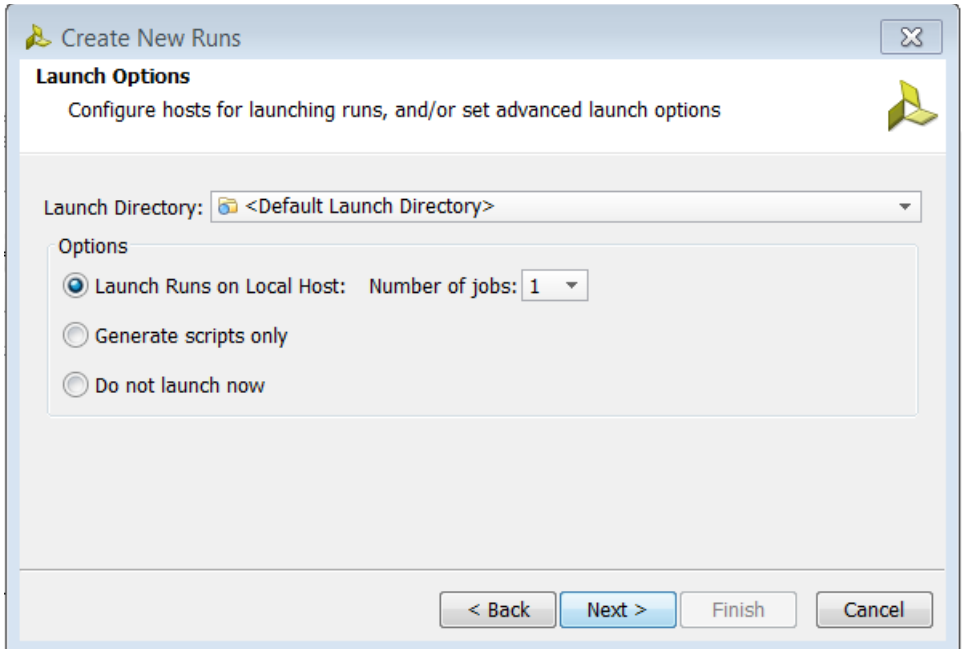

Figure 11: **Launch Options Dialog Box**

6. In the Launch Options dialog box, set the options.
   - In the **Launch Directory** drop-down menu browse to, and select the directory from which to launch the run.
   - In the **Options** area, choose one of the following:
     - **Launch Runs on Local Host**: Lets you run the options from the machine on which you are working. The **Number of jobs** drop-down lets you specify how many runs to launch.
     - **Launch Runs on Remote Hosts**: Lets you launch the runs on a remote host (Linux only) and configure that host. See "Appendix A" of the *Vivado Design Suite User Guide: Implementation (UG904)* [Ref 6], for more information about launching runs on remote hosts in Linux. The **Configure Hosts** button lets you configure the hosts from this dialog box.
     - **Generate scripts only**: Lets you generate scripts to run later. Use `runme.bat` (Windows) or `runme.sh` (Linux) to start the run.

- **Do not launch now**: Lets you save the settings that you defined in the previous dialog boxes and launch the runs at a later time.

After setting the Create New Runs wizard option and starting a run, you can see the results in the Design Runs window, as shown in Figure 12.

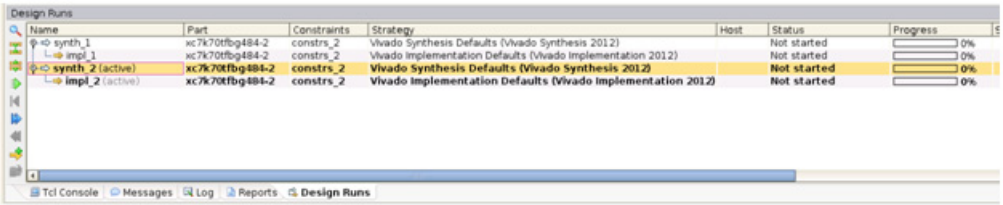

Figure 12: **Design Runs Window**

Using the Design Runs Window

The Design Runs window displays the synthesis and implementation runs created in a project and provides commands to configure, manage, and launch the runs.

If the Design Runs window is not already displayed, select **Window > Design Runs** to open the Design Runs window, shown in Figure 12.

A synthesis run can have multiple implementation runs. To expand and collapse synthesis runs, use the tree widgets in the window.

The Design Runs window reports the run status, when the run has not been started, is in progress, is complete, or is out-of-date.

Runs become out-of-date when you modify source files, constraints, or project settings. To reset or delete specific runs, right-click the run and select the appropriate command.

Setting the Active Run

Only one synthesis run and one implementation run can be *active* in the Vivado IDE at any time. All the reports and tab views display the information for the active run. The Project Summary window only displays compilations, resource, and summary information for the active run.

To make a run active, select the run in the Design Runs window and use the **Make Active** command from the popup menu to set it as the active run.

Launching a Synthesis Run

To launch a Synthesis run, do one of the following:

- From the Flow Navigator section, click the **Run Synthesis** command.
- From the main menu, select the **Flow > Run Synthesis** command.
- Right-click on the run in the Design Runs window, and select **Launch Runs**.

  The first two options start the active Synthesis run. The third option opens the Launch Selected Runs window. Here, you can select to run on local host, run on a remote host, or generate the scripts to be run.

  See "Appendix A" of the *Vivado Design Suite User Guide: Implementation (UG904)* [Ref 6], for more information about using remote hosts.

TIP: *Each time a run is launched, Vivado synthesis spawns a separate process. Be aware of this when examining messages, which are process-specific.*

Moving Processes to the Background

As the Vivado IDE initiates the process to run synthesis or implementation, an option in the dialog box lets you put the process into the background. When the run is put in the background, it releases theVivado IDE to perform other functions, such as viewing reports.

Monitoring the Synthesis Run

Monitor the status of a Synthesis run from the Log window, shown in Figure 13. The messages that show in this window during synthesis are also the messages included in the synthesis log file.

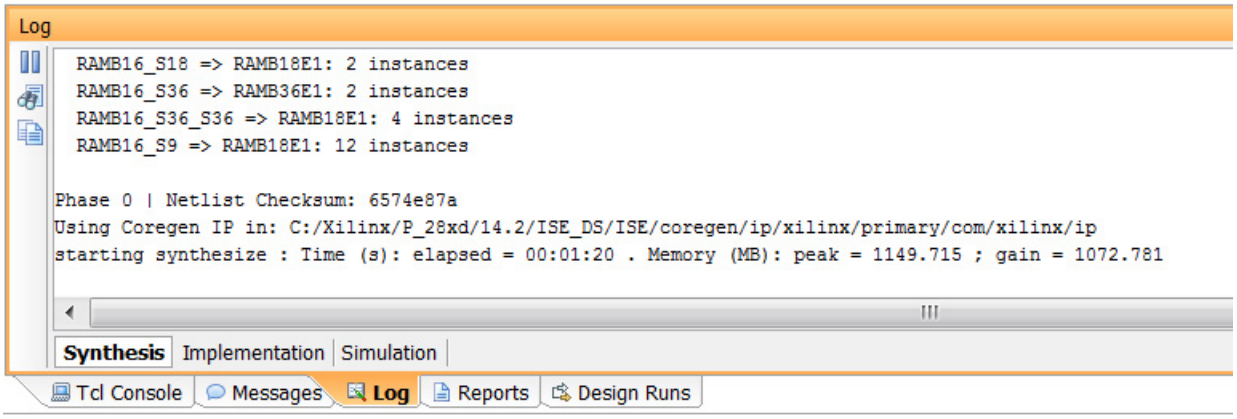

Figure 13: **Log Window**

Following Synthesis

After the run is complete, the Synthesis Completed dialog box opens, as shown in Figure 14.

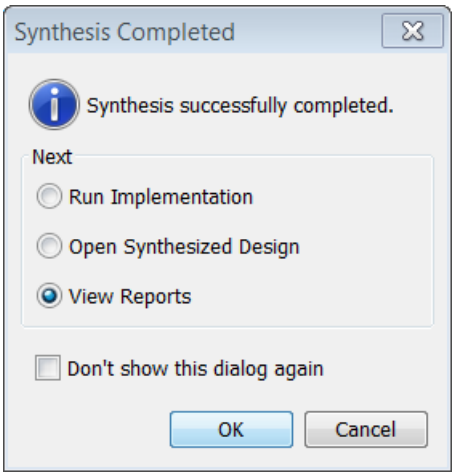

Figure 14: **Synthesis Completed Dialog Box**

Select an option:

- **Run Implementation**: Launches Implementation with the current Implementation Project Settings.
- **Open Synthesized Design**: Opens the synthesized netlist, the active constraint set, and the target device into Synthesized Design environment, so you can perform I/O pin planning, design analysis, and floorplanning.
- **View Reports:** Opens the Reports window so you can view reports.

  Use the **Don't show this dialog again** checkbox to stop this dialog box display.

TIP: *You can revert to having the dialog box present by selecting* ***Tools*** *>* ***Options*** *>* ***Windows Behavior****.*

Analyzing Synthesis Results

After Synthesis completes, you can view the reports, and open, analyze, and use the Synthesized design. The Reports window contains a list of reports provided by the Vivado IDEvarious synthesis and implementation tools.

Open the Reports window, as shown in Figure 15, and select a report for a specific run to see details of the run.

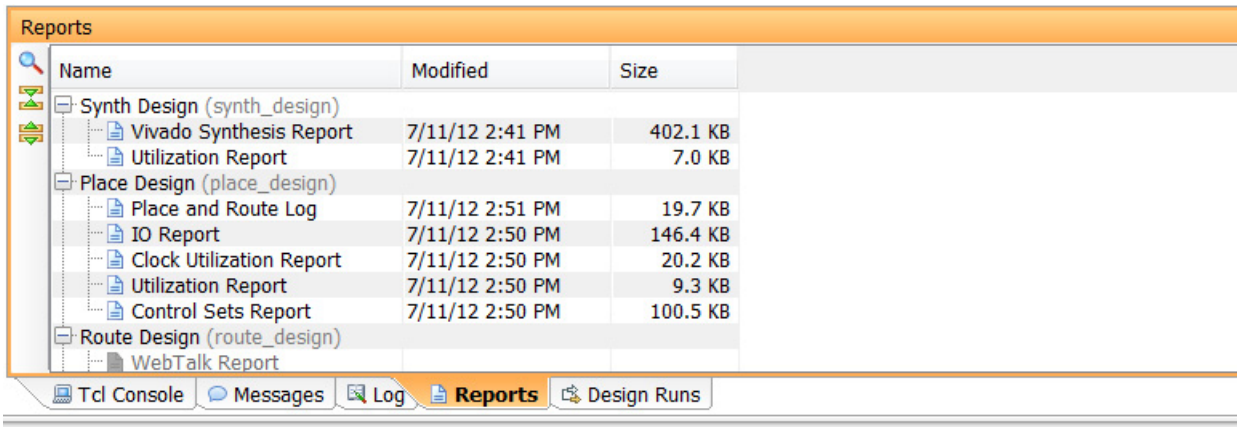

Figure 15: **Reports Window**

Using the Synthesized Design Environment

The Vivado IDE provides an environment to analyze the design from several different perspectives. When you open a synthesized design, the software loads the synthesized netlist, the active constraint set, and the target device.

See "Synthesized Design Constraints and Analysis" in the *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)* [Ref 3], for more information.

To open a synthesized design do one of the following:

- From the **Synthesis** section of the **Flow Navigator**, select **Open Synthesized Design**.
- From the main menu, select **Flow > Open Synthesized Design**.

With a Synthesized design open, the Vivado IDE opens floorplanning, as shown in Figure 16.

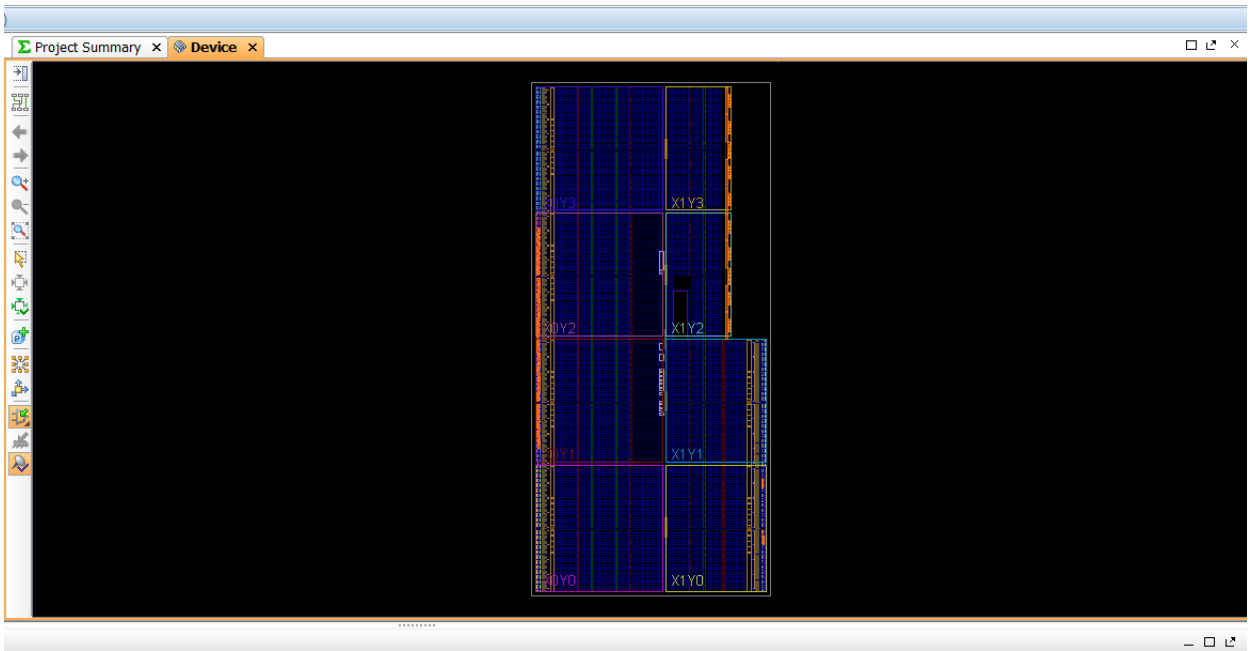

Figure 16: **Floorplanning window**

From this perspective, examine the design logic and hierarchy, view the resource utilization and timing estimates, or run Design Rule Checks (DRCs).

Viewing Floorplanning and Reporting Resource Statistics

Resource estimates display graphically as an expandable hierarchical tree. As each resources type displays, expand it to view each level of logic hierarchy.

To display a graphical view of device resource estimates, open a synthesized design by clicking either:

- **Flow Navigator > Report Utilization**
- **Tools > Report Utilization**

The Resource Utilization window opens, as shown in Figure 17.

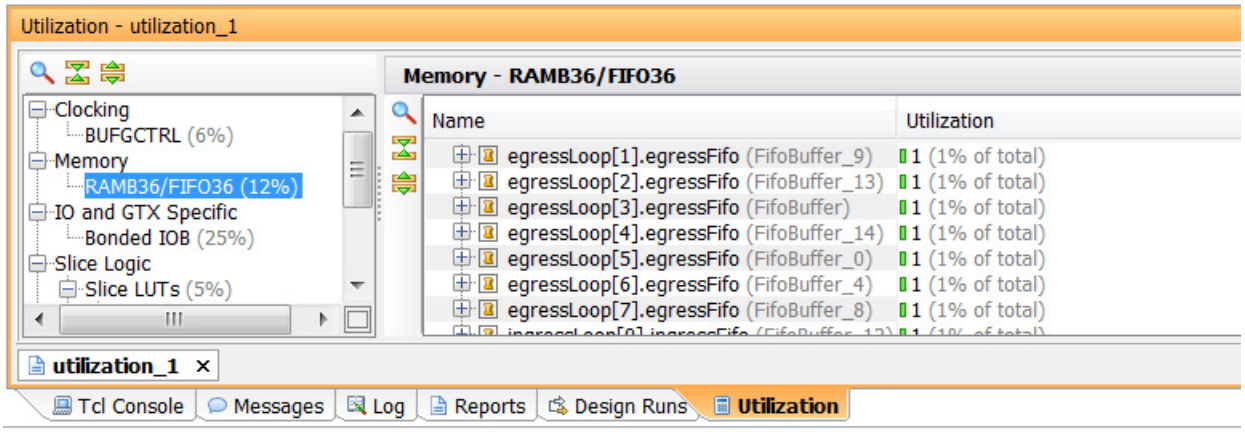

Figure 17: **Resource Utilization Window**

Viewing Resource Statistics for Logic Instances

The Vivado IDE provides estimates of the number of device resources contained in the design.

To display resource statistics for any logic instance, including the top-level, use the Instance Properties window. Select a top module or any instance module in the Netlist window, as shown in Figure 18.

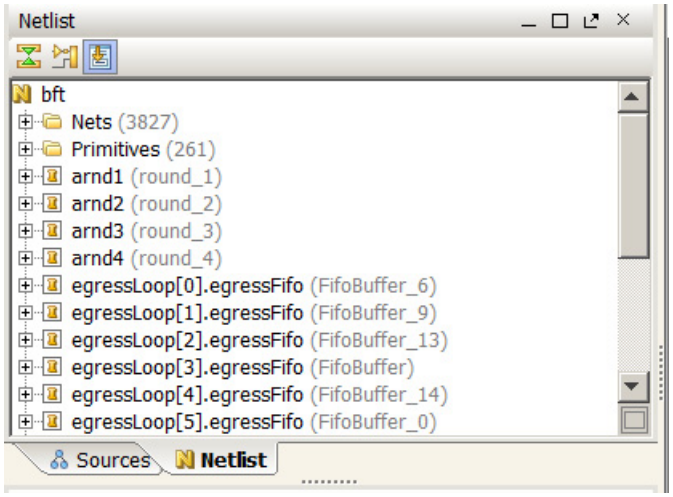

Figure 18: **Netlist Window**

If the Netlist or Instance Properties do not display, right-click on the module, and select **Netlist Properties** or **Instance Properties** from the popup menu.

In the Netlist or Instance Properties window, click the **Statistics** option, shown in Figure 19.

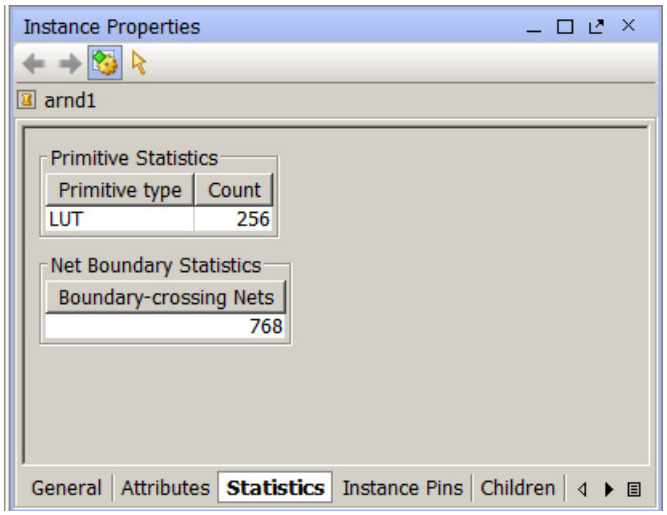

Figure 19: **Instance Properties Window**

Statistics displays design information such as: **Primitive Statistics** and **Net Boundary Statistics**.

Other options available in the Instance Properties window are:

- **General**: Provides the **Name**, **Cell**, and **Type** of the selected instance.
- **Attributes**: Lists file attributes.
- **Instance Pins**: Lists the instance pins by **ID**, **Name**, **Dir**, **BEL Pin**, and **Net**.
- **Children**: List the child instances by **ID**, **Name**, **Cell**, and **Instance Pin Count**.
- **Nets**: **ID**, **Name**, **Instance Pin,** number of **Flat Pins**, and a checkmark for the existence of a **Driver**.
- **Power**: Provides dials for percentages of **Signal Rate**, **Static Probability**, and a checkmark for **Hierarchy**.

Exporting Resource Statistics Reports

You can save displayed resource statistics data to a spreadsheet file. The Vivado IDE generates a hierarchical-style report in which you can define how many levels of hierarchy to report along with estimates listed for each module at each level.

To export a resource statistics report, in the Netlist window right-click the hierarchy on which you want statistics, and select **Export Statistics** from the popup menu, shown in Figure 20.

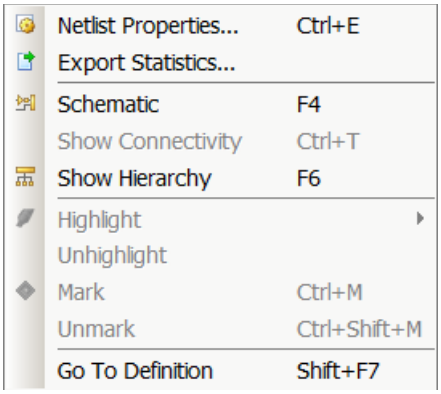

Figure 20: **Netlist Window Popup Menu**

The Export Instance Statistics dialog box opens, as shown in Figure 21.

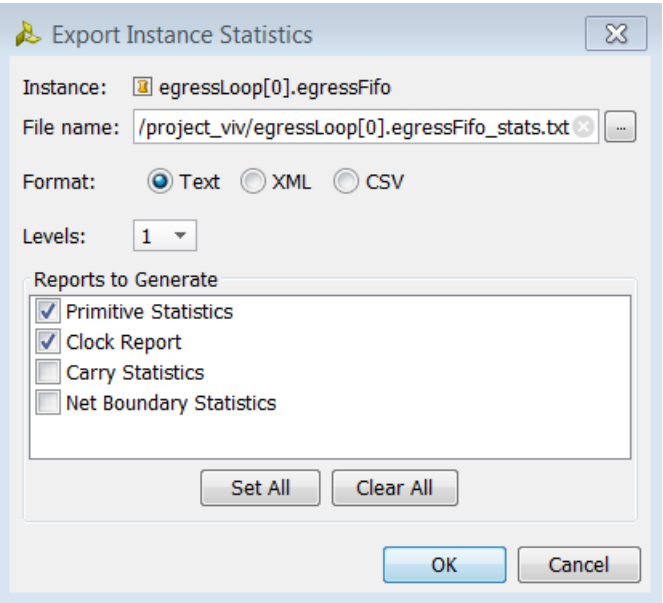

Figure 21: **Export Instance Statistics Dialog Box**

In the Export Instance Statistics dialog box provide the following information:

- **File Name**: Enter the name and location of the spreadsheet file to be created.

- **Format**: Select the format of the output file: Text, XML, or CSV.
- **Levels**: Indicate the number of levels of hierarchy to traverse and include in the report as separated modules.
- **Reports to Generate**: Define the types of netlist statistics to include in the output report file.

Select the options for the exported file, and click **OK**.

Exploring the Logic

The Vivado IDE provides several logic exploration perspectives:

- The Netlist and Hierarchy windows contain a navigable hierarchical tree-style view.
- The Schematic window allows selective logic expansion and hierarchical display.
- The Device window provides a graphical view of the device, placed logic objects, and connectivity.

All windows cross probe to present the most useful information.

Exploring the Logic Hierarchy

The Synthesized Design window displays the logic hierarchy of the synthesized design. You can expand and select any logic instance or net within the netlist.

As you select logic objects in other windows, the Netlist window expands automatically to display the selected logic objects, and the information about instances or nets displays in the Instance or Net Properties windows.

The Hierarchy window displays a graphical representation of the RTL logic hierarchy. Each module is sized in relative proportion to the others, so you can determine the size and location of any selected module.

Exploring the Logical Schematic

The Schematic window allows selective expansion and exploration of the logical design. You must select at least one logic object to open and display the Schematic window.

In the Schematic window, view and select any logic. You can display groups of timing paths to show all of the instances on the paths. This aids floorplanning because it helps you visualize where the timing critical modules are in the design.

To open the Schematic window:

1. Select one or more instances, nets, or timing paths.
2. Select **Schematic** from the window toolbar or the popup menu, or press the **F4** key.

   The window opens with the selected logic displayed, as shown in Figure 22.

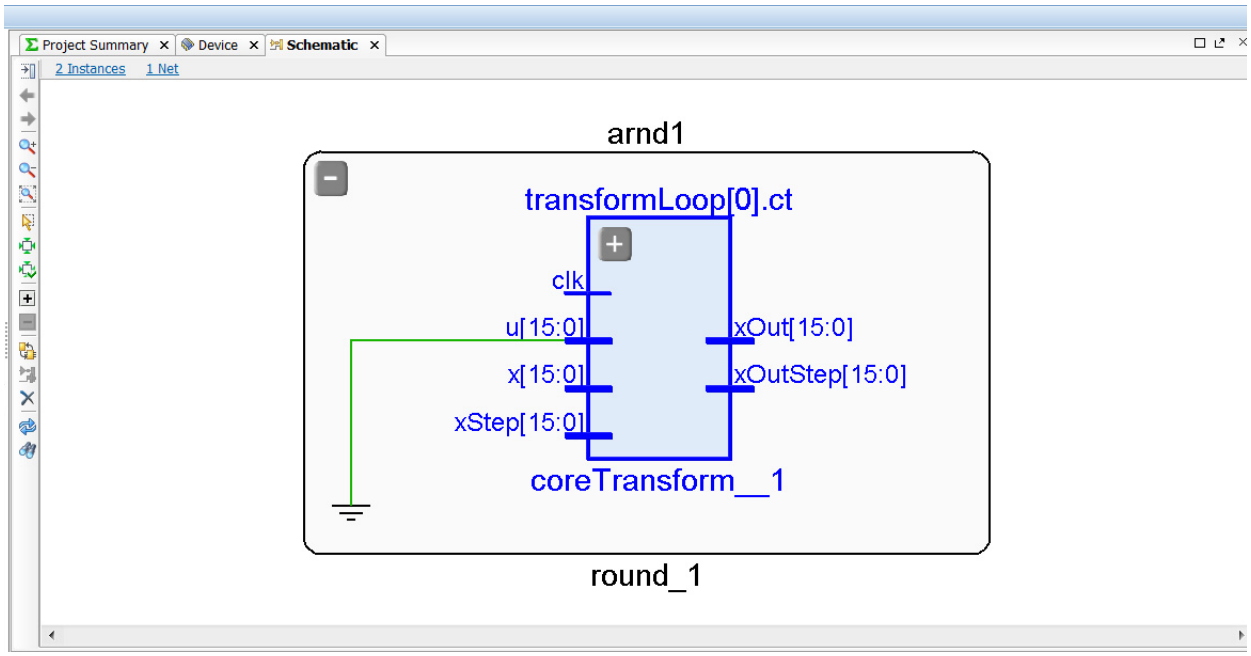

Figure 22: **Schematic Window**

You can then select and expand the logic for any pin, instance, or hierarchical module.

Running Timing Analysis

Timing analysis of the synthesized design is useful to ensure that paths have the necessary constraints for effective implementation The Vivado synthesis is timing-driven and adjusts its outputs based on provided constraints.

As more physical constraints, such as `Pblocks` and `LOC` constraints, are assigned in the design, the results of the timing analysis become more accurate, although these results still contain some estimation of path delay. The synthesized design uses an estimate of routing delay to perform analysis.

You can run timing analysis at this level to ensure that the correct paths are covered and for a more general idea of timing paths.

IMPORTANT: Only timing analysis includes the actual delays for routing. Running timing analysis on the synthesized design is not as accurate as running timing analysis on an implemented design.

Using the Report Timing Command

To perform a timing analysis, use one of the following methods:

- From the main menu, select **Tools > Timing > Report Timing Summary**.
- From the Flow Navigator Synthesized Design section, select **Report Timing Summary**.

The Report Timing Summary dialog box opens, as shown in Figure 23.

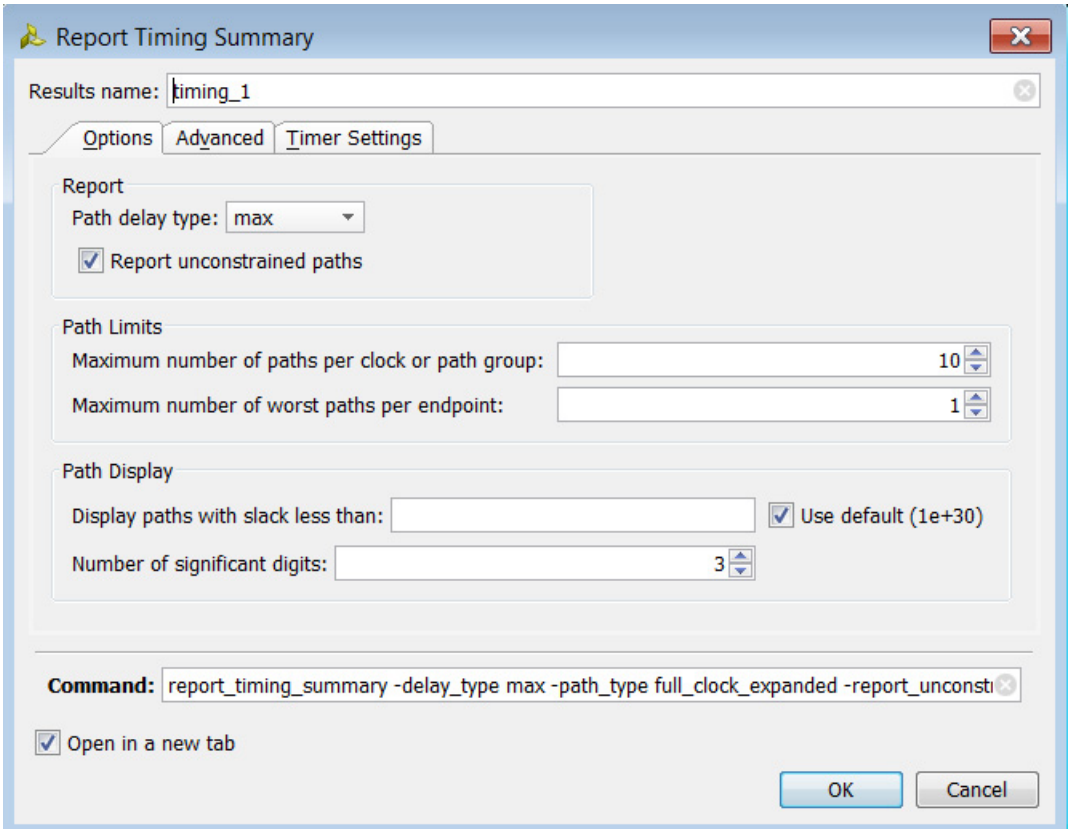

Figure 23: **Report Timing Summary Dialog Box**

Select the options as follows:

- **Results name**: Lets you name the report results.
- **Report**: Gives the option of setting the Path delay type to **Max**, **Min**, or **Min_Max** and provides a checkbox option to **Report unconstrained paths**.
- **Path Limits**: Selection options are:
  - **Maximum number of paths per clock or path group**
  - **Maximum number of worst paths per endpoint**
- **Command**: displays the current `report_timing` command encompassing the selected options, and provides a checkbox to **Open in a new tab**.

About XST Strategies

The Vivado Synthesis Defaults is the recommended strategy. If you want to target XST, you can use a predefined XST strategy. Two of the main XST strategies are:

- PlanAhead Defaults: Provides a rebuilt hierarchical netlist.
- XST Defaults: Turns off the hierarchy recovery.

Figure 24 shows the **XST 14 > PlanAhead Defaults** and **XST Defaults** in the Vivado Strategies tree.

Figure 24: **Vivado XST Default Strategy**

For more information about XST options and reports, see the following:

- *PlanAhead User Guide (UG632)* [Ref 9]
- *XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices (UG687)* [Ref 10]

Using Non-Project Mode for Synthesis

The Tcl command to run synthesis is `synth_design`. Typically, this command is run with multiple options, for example:

```
synth_design -part xc7k30tfbg484-2 -top my_top
```

In this example, `synth_design` is run with the `-part` option and the `-top` option.

In the Tcl Console, you can set synthesis options and run synthesis using Tcl command options. To retrieve a list of options, type `synth_design -help` in the Tcl Console. The following snippet is an example of the `-help` output:

```
synth_design -help
synth_design
Description:
Synthesize a design using Vivado Synthesis and open that design
Syntax:
synth_design  [-name <arg>] [-part <arg>] [-constrset <arg>] [-top <arg>]
              [-include_dirs <args>] [-generic <args>] [-verilog_define <args>]
              [-flatten_hierarchy <arg>] [-gated_clock_conversion <arg>]
              [-effort_level <arg>] [-rtl] [-no_iobuf] [-bufg <arg>]
              [-fanout_limit <arg>] [-fsm_extraction <arg>] [-quiet] [-verbose]
Returns:
design object
Usage:
  Name                       Description
  -------------------------------------
  [-name]                    Design name
  [-part]                    Target part
  [-constrset]               Constraint fileset to use
  [-top]                     Specify the top module name
  [-include_dirs]            Specify verilog search directories
  [-generic]                 Specify generic parameters. Syntax: -generic
                             <name>=<value> -generic <name>=<value> ...
  [-verilog_define]          Specify verilog defines. Syntax: -verilog_define
                             <macro_name>[=<macro_text>] -verilog_define
                             <macro_name>[=<macro_text>] ...
  [-flatten_hierarchy]       Flatten hierarchy during LUT mapping. Values:
                             full, none, rebuilt
                             Default: rebuilt
  [-gated_clock_conversion]  Convert clock gating logic to flop enable.
                             Values: off, on, auto Default: off
  [-effort_level]            Synthesis effort level. Values: quick, med
                             Default: med
  [-rtl]                     Elaborate and open an rtl design
  [-no_iobuf]                Disable setting of I/O buffers
  [-bufg]                    Max number of global clock buffers used by
                             synthesis.Default: 12
  [-fanout_limit]            Fanout limit Default: 100
[-fsm_extraction]            FSM Extraction Encoding. Values: off, one_hot,
                             sequential, johnson, gray, auto. Default: off
  [-quiet]                   Ignore command errors
  [-verbose]                 Suspend message limits during command execution
```

A verbose version of the help is available in the *Vivado Tcl Command Reference Guide (UG835)* [Ref 4]. To determine any Tcl equivalent to a Vivado IDE action, run the command in the Vivado IDE and review the content in the TCL Console or the log file.

The following is an example `synth_design` Tcl script:

```
# Setup design sources and constraints
read_vhdl -library bftLib [ glob ./Sources/hdl/bftLib/*.vhdl ]
read_vhdl ./Sources/hdl/bft.vhdl
read_verilog [ glob ./Sources/hdl/*.v ]
read_xdc ./Sources/bft_full.xdc
# Run synthesis, report utilization and timing estimates, write design checkpoint
synth_design -top bft -part xc7k70tfbg484-2 -flatten rebuilt
write_checkpoint -force $outputDir/post_synth
```

Setting Constraints

Table 1 shows the supported Tcl commands for Vivado timing constraints.

For details on these commands, see the following documents:

- "Synthesized Design Constraints and Analysis" in the *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)* [Ref 3]
- *Vivado Tcl Command Reference Guide (UG835) [Ref 4]*

Table 1: **Supported Synthesis Tcl Commands**

Command Type	Commands			
Timing Constraints	`create_clock`	`create_generate_clock`	`set_false_path`	`set_input_delay`
	`set_output_delay`	`set_max_delay`	`set_multicycle_path`	
	`set_clock_latency`	`set_clock_groups`	`set_disable_timing`	
Object Access	`all_clocks`	`all_inputs`	`all_outputs`	`get_cells`
	`get_clocks`	`get_nets`	`get_pins`	`get_ports`

Synthesis Attributes

Introduction

In the Vivado™ Design Suite, Vivado synthesis is able to use many of the XST synthesis attributes. In most cases, these attributes have the same syntax and the same behavior.

- If Vivado synthesis supports the attribute, it uses the attribute, and creates logic that reflects the used attribute.
- If the specified attribute is not recognized by the tool, the Vivado synthesis passes the attribute and its value to the generated netlist.

It is assumed that a tool downstream in the flow can use the attribute. For example, the `LOC` constraint is not used by synthesis, but the constraint is used by the Vivado placer, and is forwarded by Vivado synthesis.

Supported Attributes

The following subsections list the supported attributes.

BLACK_BOX

The `BLACK_BOX` attribute is a useful debugging attribute that can turn a whole level of hierarchy off and enable synthesis to create a black box for that module or entity. When the attribute is found, even if there is valid logic for a module or entity, Vivado synthesis creates a black box for that level.

BLACK_BOX Verilog Example

```
(* black_box *) module test(in1, in2, clk, out1);
```

BLACK_BOX VHDL Example

```
attribute black_box  : string;
attribute black_box  of beh : architecture is "yes";
```

In the Verilog example, no value is needed. The presence of the attribute creates the black box.

BUFFER_TYPE

Apply `BUFFER_TYPE` on an input to describe what type of buffer to use.

By default, Vivado synthesis uses `IBUF/BUFG` or `BUFGP`s for clocks and `IBUF`s for inputs.

Supported values are:

- `ibuf`: For clock ports where a `IBUF/BUFG` pair is not wanted. In this case only, the `IBUF` is inferred for the clock.
- `none`: Indicates that no input or output buffers are used. A `none` value on a clock port results in no buffers.

Note: XST supports other values such as "`ibufg`", "`bufr`", "`bufgp`", and "`bufg`".

BUFFER_TYPE Verilog Example

```
(* buffer_type = "none" *) input in1;  //this will result in no buffers
(* buffer_type = "ibuf" *) input clk1;  //this will result in a clock with no bufg
```

BUFFER_TYPE VHDL Example

```
entity test is port(
in1 : std_logic_vector (8 downto 0);
clk : std_logic;
out1 : std_logic_vector(8 downto 0));
attribute buffer_type : string;
attribute buffer_type of in1 : signal is "none";
end test;
```

DONT_TOUCH

Use DONT_TOUCH attribute in place of KEEP or KEEP_HIERARCHY and works in the same way as these attributes. However, unlike KEEP and KEEP_HIERARCHY, DONT_TOUCH is forward annotated to place and route to prevent logic optimization.

Note: Replace KEEP and KEEP_HIERARCHY attributes with DONT_TOUCH.

DONT_TOUCH Verilog Example

```
(* dont_touch = "true" *) wire sig1;
assign sig1 = in1 & in2;
assign out1 = sig1 & in2;
```

DONT_TOUCH VHDL Example

```
signal sig1 : std_logic
attribute dont_touch : string;
attribute dont_touch of sig1 : signal is "true";
....
....
sig1 <= in1 and in2;
out1 <= sig1 and in3;
```

FULL_CASE (Verilog Only)

FULL_CASE indicates that all possible case values are specified in a case, casex, or casez statement. If case values are specified, extra logic for case values is not created by Vivado synthesis.

```
(* full_case *)
case select
3'b100 : sig = val1;
3'b010 : sig = val2;
3'b001 : sig = val3;
endcase
```

IMPORTANT: *This attribute can only be controlled through RTL.*

GATED CLOCK

Vivado synthesis allows the conversion of gated clocks. The two items to use to perform this conversion are:

- A switch in the UI, that instructs the tool to attempt the conversion.
- The RTL attribute that instructs the tool about which signal in the gated logic is the clock.

To control the switch:

1. In the Flow Navigator, go to Synthesis Settings section.
2. In the Options window, set the `-gated_clock_conversion` option to one of the following values:
   - `off`: Disables the gated clock conversion.
   - `on`: Gated clock conversion occurs if the `gated_clock` attribute is set in the RTL code. This option gives you more control of the outcome.
   - `auto`: Gated clock conversion occurs if either of the following events are true:
     - the `gated_clock` attribute is set to `true`
     - the Vivado synthesis can detect the gate and there is a valid clock constraint set

     This option lets the tool make decisions.

GATED_CLOCK Verilog Example

```
(* gated_clk = "true" *) input clk;
GATED_CLOCK VHDL example:
entity test is port (
in1, in2 : in std_logic_vector(9 downto 0);
en : in std_logic;
clk : in std_logic;
out1 : out std_logic_vector( 9 downto 0));
attribute gated_clock : string;
attribute gated_clock of clk : signal is "true";
end test;
```

KEEP

Use the `KEEP` attribute to prevent optimizations where signals are either optimized or absorbed into logic blocks. This attribute instructs the synthesis tool to keep the signal it was placed on, and that signal is placed in the netlist. For example, if a signal is an output of a 2 bit `AND` gate, and it drives another `AND` gate, the `KEEP` attribute can be used to prevent that signal from being merged into a larger `LUT` that encompasses both `AND` gates.

KEEP is also commonly used in conjunction with timing constraints. If there is a timing constraint on a signal that would normally be optimized, KEEP prevents that and allows the correct timing rules to be used.

The supported KEEP values are:

- true: Keeps the signal.
- false: Allows the Vivado synthesis to optimize, if it determines that it should. false does not force the tool to remove the signal. The default value is false.

Note: The KEEP attribute does not force the place and route to keep the signal. Instead, this is accomplished using the DONT_TOUCH attribute.

KEEP Verilog Example

```
(* keep = "true" *) wire sig1;
assign sig1 = in1 & in2;
assign out1 = sig1 & in2;
```

KEEP VHDL Example

```
signal sig1 : std_logic
attribute keep : string;
attribute keep of sig1 : signal is "true";
....
....
sig1 <= in1 and in2;
out1 <= sig1 and in3;
```

KEEP_HIERARCHY

KEEP_HIERARCHY is used to prevent optimizations along the hierarchy boundaries. The Vivado synthesis tool attempts to keep the same general hierarchies specified in the RTL, but for QoR reasons it can flatten or modify them.

If KEEP_HIERARCHY is placed on the instance, the synthesis tool keeps the boundary on that level static. This can affect QoR and also should not be used on modules that describe the control logic of tristate outputs and I/O buffers. The KEEP_HIERARCHY can be placed in the module or architecture level or the instance.

KEEP_HIERARCHY Verilog Example

On Module:

```
(* keep_hierarchy = "yes" *) module bottom (in1, in2, in3, in4, out1, out2);
```

On Instance:

```
(* keep_hierarchy = "yes" *)bottom u0 (.in1(in1), .in2(in2), .out1(temp1));
```

KEEP_HIERARCHY VHDL Example

On Module:

```
attribute keep_hierarchy : string;
attribute heep_hierarchy of beh : architecture is "yes";
```

On Instance:

```
attribute keep_hierarchy : string;
attribute keep_hierarchy of u0 : label is "yes";
```

MAX_FANOUT

`MAX_FANOUT` instructs Vivado synthesis on the fanout limits for registers and signals. You must also place a `KEEP` attribute on the signal. You can specify this either in RTL or as an input to the project. The value is an integer.

- If the attribute is specified in the RTL, this is a hard limit.
- If `MAX_FANOUT` is specified in the project, it is a soft limit. A soft limit means that the tool attempts to keep the limit, but it can ignore the limit if this provides better results.

  This attribute only works on registers and combinatorial signals. To achieve the fanout, it replicates the register or the driver that drives the combinatorial signal.

  Note: Inputs and black boxes are not currently supported.

MAX_FANOUT Verilog Example

In Vivado:

```
(* keep = "true", max_fanout = 50 *) reg sig1;
```

In XST:

```
(* keep = "true", max_fanout = "50" *) reg sig1;
```

MAX_FANOUT VHDL Example

```
signal sig1 : std_logic;
attribute keep : string;
attribute max_fanout : integer;
attribute keep of sig1 : signal is "true";
attribute max_fanout : signal is 50;
```

- In VHDL Vivado synthesis, `max_fanout` is an `integer`.
- In VHDL XST, it is a `string`.

PARALLEL_CASE (Verilog Only)

PARALLEL_CASE specifies that the case statement must be built as a parallel structure. Logic is not created for an if-elsif structure.

```
 (* parallel_case *)  case select
3'b100 : sig = val1;
3'b010 : sig = val2;
3'b001 : sig = val3;
endcase
```

IMPORTANT: *This attribute can only be controlled through the Verilog RTL.*

RAM_STYLE

RAM_STYLE instructs the Vivado synthesis tool on how to infer memory. Accepted values accepted are:

- block: Instructs the tool to infer RAMB type components
- distributed: Instructs the tool to infer the LUT RAMs.

  By default, the tool selects which RAM to infer based upon heuristics that give the best results for the most designs.

RAM_STYLE Verilog Example

```
(* ram_style = "distributed" *) reg [data_size-1:0] myram [2**addr_size-1:0];
```

RAM_STYLE VHDL Example

```
attribute ram_style : string;
attribute ram_style of myram : signal is "distributed";
```

ROM_STYLE

ROM_STYLE instructs the synthesis tool how to infer ROM memory. Accepted values accepted are

- block: Instructs the tool to infer RAMB type components
- distributed: Instructs the tool to infer the LUT ROMs. By default, the tool selects which ROM to infer based on heuristics that give the best results for the most designs.

ROM_STYLE Verilog Example

```
(* rom_style = "distributed" *) reg [data_size-1:0] myrom [2**addr_size-1:0];
```

ROM_STYLE VHDL Example

```
attribute rom_style : string;
attribute rom_style of myrom : signal is "distributed";
```

TRANSLATE_OFF/TRANSLATE_ON

`TRANSLATE_OFF` and `TRANSLATE_ON` instruct the Synthesis tool to ignore blocks of code. These attributes are given within a comment in RTL. The comment should start with one of the following keywords:

- `synthesis`
- `synopsys`
- `pragma`

  `TRANSLATE_OFF` starts the ignore, and it ends with `TRANSLATE_ON`. These commands cannot be nested.

TRANSLATE_OFF/TRANSLATE_ON Verilog Example

```
// synthesis translate_off
Code....
// synthesis translate_on
```

TRANSLATE_OFF/TRANSLATE_ON VHDL Example

```
-- synthesis translate_off
Code...
-- synthesis translate_on
```

CAUTION! *Be careful with the types of code that are included between the translate statements. If it is code that affects the behavior of the design, a simulator could use that code, and create a simulation mismatch.*

USE_DSP48

`USE_DSP48` instructs the synthesis tool how to deal with synthesis arithmetic structures. By default, `mults`, `mult-add`, `mult-sub`, `mult-accumulate` type structures go into DSP48 blocks. Adders, subtractors, and accumulators can also go into these blocks but by default, are implemented with the fabric instead of with DSP48 blocks. The `USE_DSP48` attribute overrides the default behavior and force these structures into DSP48 blocks.

Accepted values are `"yes"` and `"no"`. This attribute can be placed in the RTL on signals, architectures and components, entities and modules. The priority is as follows:

1. Signals
2. architectures and components
3. modules and entities.

If the attribute is not specified, the default behavior is for Vivado synthesis to determine the correct behavior.

USE_DSP48 Verilog Example

```
(* use_dsp48 = "yes" *) module test(clk, in1, in2, out1);
```

USE_DSP48 VHDL Example

```
attribute use_dsp48 : string
attribute use_dsp48 of P_reg : signal is "no"
```

Appendix B

SystemVerilog Support

Introduction

Vivado™ synthesis supports the synthesizable subset of the SystemVerilog RTL described in the following sections.

Targeting SystemVerilog for a Specific File

By default, the Vivado synthesis tool compiles `*.v` files with the Verilog 2001 syntax and `*.sv` files with the SystemVerilog syntax.

To target SystemVerilog for a specific `*.v` file in the Vivado IDE:

1. Right-click the file, and select **Source Node Properties**.
2. In the Source Node Properties window, change the Type from **Verilog** to **SystemVerilog**, and click **Apply**.

   Alternatively, you can use the following Tcl command in the Tcl Console:

   ```
   set_property file_type SystemVerilog [get_files <filename>.v]
   ```

The following sections describe the supported SystemVerilog types in the Vivado IDE.

Data Types

The following data types are supported, as well as the mechanisms to control them.

Declaration

Declare variables in the RTL as follows:

```
[var] [DataType] name;
```

Where:

- `Var` is optional and implied if not in the declaration.
- `DataType` is one of the following:
  - `integer_vector_type: bit`, `logic`, or `reg`
  - `integer_atom_type: byte`, `shortint`, `int`, `longint`, `integer`, or `time`
  - `non_integer_type: shortreal`, `real`, or `realtime`
  - `struct`
  - `enum`

Integer Data Types

SystemVerilog supports the following integer types:

- `shortint`: 2-state 16-bit signed integer
- `int`: 2-state 32-bit signed integer
- `longint`: 2-state 64-bit signed integer
- `byte`: 2-state 8-bit signed integer
- `bit`: 2-state, user defined vector size
- `logic`: 4-state user defined vector size
- `reg`: 4-state user-defined vector size
- `integer`: 4-state 32-bit signed integer
- `time`: 4-state 64-bit unsigned integer

4-state and 2-state refer to the values that can be assigned to those types, as follows:

- 2-state allows 0s and 1s.
- 4-state also allows X and Z states.

X and Z states cannot always be synthesized; therefore, items that are 2-state and 4-state are synthesized in the same way.

CAUTION! *Take care when using 4-state variables: RTL versus simulation mismatches could occur.*

- The types `byte`, `shortint`, `int`, `integer`, and `longint` default to signed values.
- The types `bit`, `reg`, and `logic` default to unsigned values.

Real Numbers

Synthesis supports real numbers; however, they cannot be used for behavior. They can be used as parameter values. The SystemVerilog-supported real types are:

- `real`
- `shortreal`
- `realtime`

Void Data Type

The `void` data type is only supported for functions that have no return value.

User-Defined Types

Vivado synthesis supports user-defined types, which are defined using the `typedef` keyword. Use the following syntax:

```
typedef data_type type_identifier {size};
```

or

```
typedef [enum, struct, union] type_identifier;
```

Enum Types

Enumerated types can be declared with the following syntax:

```
enum [type] {enum_name1, enum_name2...enum_namex} identifier
```

If no type is specified, the `enum` defaults to `int`. Following is an example:

```
enum {sun, mon, tues, wed, thurs, fri, sat} day_of_week;
```

This code generates an `enum` of `int` with seven values. The values that are given to these names start with 0 and increment, so that, `sun = 0` and `sat = 6`.

To override the default values, use code as in the following example:

```
enum {sun=1, mon, tues, wed, thurs, fri, sat} day_of week;
```

In this case, `sun` is 1 and `sat` is 7.

The following is another example how to override defaults:

```
enum {sun, mon=3, tues, wed, thurs=10, fri=12, sat} day_of_week;
```

In this case, `sun=0`, `mon=3`, `tues=4`, `wed=5`, `thurs=10`, `fri=12`, and `sat=13`.

Enumerated types can also be used with the `typedef` keyword.

```
typedef enum {sun,mon,tues,wed,thurs,fri,sat} day_of_week;
day_of_week my_day;
```

The preceding example defines a signal called `my_day` that is of type `day_of_week`. You can also specify a range of `enums`. For example, the preceding example can be specified as:

```
enum {day[7]} day_of_week;
```

This creates an enumerated type called `day_of_week` with N-1 elements called `day0`, `day1`…`day6`.

Following are other ways to use this:

```
enum {day[1:7]} day_of_week;  // creates day1,day2...day7
enum {day[7] = 5} day_of_week; //creates day0=5, day1=6... day6=11
```

Constants

SystemVerilog gives three types of elaboration-time constants:

- `parameter`: Is the same as the original Verilog standard and can be used in the same way.
- `localparam`: Is similar to `parameter` but cannot be overridden by upper-level modules.
- `specparam`: Is used for specifying delay and timing values; consequently, this value is *not supported* in Vivado synthesis.

There is also a run-time constant declaration called `const`.

Type Operator

The type operator allows parameters to be specified as data types, which allows modules to have different types of parameters for different instances.

Casting

Assigning a value of one data type to a different data type is illegal in SystemVerilog. However, a workaround is to use the cast operator (`'`). The cast operator converts the data type when assigning between different types. The usage is:

```
casting_type'(expression)
```

The `casting_type` is one of the following:

- `integer_type`
- `non_integer_type`
- `real_type`
- constant unsigned number
- user-created signing value type

Aggregate Data Types

In aggregate data types there are *structures* and *unions*, which are described in the following subsections.

Structures

A structure is a collection of data that can be referenced as one value, or the individual members of the structure. This is similar to the VHDL concept of a record. The format for specifying a structure is:

```
struct {struct_member1; struct_member2;...struct_memberx;} structure_name;
```

Unions

A union is a data type comprising multiple data types. Only one data type is used. This is useful in cases where the data type changes depending on how it is used. The following code snippet is an example:

```
typedef union {int i; logic [7:0] j} my_union;
my_union sig1;
my_union sig2;
sig1.i = 32;  //sig1 will get the int format
sig2.j = 8'b00001111;  //sig2 will get the 8bit logic format.
```

Packed and Unpacked Arrays

Vivado synthesis supports both packed and unpacked arrays:

```
logic [5:0] sig1;  //packed array
logic sig2 [5:0]; //unpacked array
```

Data types with predetermined widths do not need the packed dimensions declared:

```
integer sig3;  //equivalent to logic signed [31:0] sig3
```

Processes

Always Procedures

There are four `always` procedures:

- `always`
- `always_comb`
- `always_latch`
- `always_ff`

  The procedure `always_comb` describes combinational logic. A sensitivity list is inferred by the logic driving the `always_comb` statement.

  For `always` you must provide the sensitivity list. The following examples use a sensitivity list of `in1` and `in2`:

```
always@(in1 or in2)
out1 = in1 & in2;
always_comb out1 = in1 & in2;
```

  The procedure `always_latch` provides a quick way to create a latch. Like `always_comb`, a sensitivity list is inferred, but you must specify a control signal for the latch enable, as in the following example:

```
always_latch
 if(gate_en) q <= d;
```

  The procedure `always_ff` is a way to create flip-flops. Again, you must specify a sensitivity list:

```
always_ff@(posedge clk)
out1 <= in1;
```

Block Statements

Block statements provide a mechanism to group sets of statements together. Sequential blocks have a `begin` and `end` around the statement. The block can declare its own variables, and those variables are specific to that block. The sequential block can also have a name associated with that block. The format is as follows:

```
begin [: block name]
[declarations]
[statements]
end [: block name]
```

```
begin : my_block
logic temp;
temp = in1 & in2;
out1 = temp;
end : my_block
```

In the previous example, the block name is also specified after the `end` statement. This makes the code more readable, but it is not required.

Note: Parallel blocks (or fork join blocks) are *not* supported in Vivado synthesis.

Procedural Timing Controls

SystemVerilog has two types of timing controls:

- **Delay control**: Specifies the amount of time between the statement its execution. This is not useful for synthesis, and Vivado synthesis ignores the time statement while still creating logic for the assignment.
- **Event control**: Makes the assignment occur with a specific event; for example, `always@(posedge clk)`. This is standard with Verilog, but SystemVerilog includes extra functions.

  The logical `or` operator is an ability to give any number of events so that any one of them will trigger the execution of the statement. To do this, use either a specific `or`, or separate with commas in the sensitivity list. For example, the following two statements are the same:

  ```
  always@(a or b or c)
  always@(a,b,c)
  ```

  SystemVerilog also supports the implicit `event_expression @*`. This helps to eliminate simulation mismatches caused because of incorrect sensitivity lists, for example:

  ```
  Logic always@* begin
  ```

Operators

Vivado synthesis supports the following SystemVerilog operators:

- Assignment operators
  (`=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, <<<=, >>>=`)
- Unary operators (`+, -, !, ~, &, ~&, |, ~|, ^, ~^, ^~`)
- Increment/decrement operators (`++, --`)

- Binary operators (`+, -, *, /, %, ==, ~=, ===, ~==, &&, ||, **, < , <=, >, >=, &, |, ^, ^~, ~^, >>, <<, >>>, <<<`)

  Note: `A**B` is supported if `A` is a power of `2` or `B` is a constant.
- Conditional operator (`? :`)
- Concatenation operator (`{…}`)

Signed Expressions

Vivado synthesis supports both signed and unsigned operations. Signals can be declared as unsigned or signed. For example:

```
logic [5:0] reg1;
logic signed [5:0] reg2;
```

Procedural Programming Assignments

Conditional if-else Statement

The syntax for a conditional `if-else` statement is:

```
if (expression)
  command1;
else
  command2;
```

The `else` is optional and assumes a latch or flip-flop depending on whether or not there was a clock statement. Code with multiple `if` and `else` entries can also be supported, as shown in the following example:

```
If (expression1)
Command1;
else if (expression2)
command2;
else if (expression3)
command3;
else
command4;
```

This example is synthesized as a priority `if` statement. So, if the first expression is found to be `true`, the others are not evaluated. If unique or priority `if-else` statements are used, Vivado synthesis treats those as `parallel_case` and `full_case` respectively.

Case Statement

The syntax for a `case` statement is:

```
case (expression)
value1: statement1;
value2: statement2;
value3: statement3;
default: statement4;
endcase
```

The default statement inside a `case` statement is optional. The values are evaluated in order, so if both `value1` and `value3` are true, `statement1` is performed.

In addition to `case`, there are also the `casex` and `casez` statements. These allow you to `handle don't cares` in the values (`casex`) or tristate conditions in the values (`casez`).

If unique or priority case statements are used, Vivado synthesis treats those as `parallel_case` and `full_case` respectively.

Loop Statements

Several types of loops that are supported in Vivado synthesis and SystemVerilog. One of the most common is the `for` loop. Following is the syntax:

```
for (initialization; expression; step)
statement;
```

A `for` loop starts with the initialization, then evaluates the expression. If the expression evaluates to 0, it stops, else if the expression evaluates to 1, it continues with the statement. When it is done with the statement, it executes the step function.

- A `repeat` loop works by performing a function a stated number of times. Following is the syntax:

  ```
  repeat (expression)
    statement;
  ```

  This works by evaluating the expression to a number and then executing the statement that many times.

- The `for-each` loop executes a statement for each element in an array
- The `while` loop takes an expression and a statement and executes the statement until the expression is `false`.
- The `do-while` loop performs the same function as the while loop, but instead it tests the expression after the statement.
- The `forever` loop executes all the time. To avoid infinite loops, use it with the break statement to get out of the loop.

Tasks and Functions

Tasks

The syntax for a `task` declaration is:

```
task name (ports);
[optional declarations];
statements;
endtask
```

Following are the two types of tasks:

- **Static task**: Declarations retain their previous values the next time the task is called.
- **Automatic task**: Declarations do not retain previous values.

CAUTION! *Be careful when using these tasks; Vivado synthesis treats all tasks as automatic.*

Many simulators default to static tasks if the static or automatic is not specified, so there is a chance of simulation mismatches. The way to specify a task as automatic or static is the following:

```
task automatic my_mult... //or
task static my_mult ...
```

Functions (Automatic and Static)

Functions are similar to tasks, but return a value. The format for a task is:

```
function data_type function_name(inputs);
  declarations;
  statements;
endfunction : function_name
```

The final `function_name` is optional but does make the code easier to read. Because the function returns a value, it must either have a return statement or specifically say in the statement:

```
function_name = ....
```

Like tasks, functions can also be automatic or static. Vivado synthesis treats all functions as automatic. However, some simulators might behave differently. Be careful when using these functions.

Modules and Hierarchy

Using modules in SystemVerilog is very similar to Verilog, and includes additional features as described in the following subsections.

Connecting Modules

There are three main ways to instantiate and connect modules. The first two are by ordered list and by name, as in Verilog. The third is by named ports.

If the names of the ports of a module match the names and types of signals in an instantiating module, the lower-level module can by hooked up by name. For example:

```
module lower (
output [4:0] myout;
input clk;
input my_in;
input [1:0] my_in2;
... ...
endmodule
//in the instantiating level.
lower my_inst (.myout, .clk, .my_in, .my_in2);
```

Connecting Modules with Wildcard Ports

You can use wildcards when hooking up modules. For example, from the previous example:

```
// in the instantiating module
lower my_inst (.*);
```

This hooks up the entire instance, as long as the upper-level module has the correct names and types.

In addition, these can be mixed and matched. For example:

```
lower my_inst (.myout(my_sig), .my_in(din), .*);
```

This hooks up the `myout` port to a signal called `my_sig`, the `my_in` port to a signal called `din` and `clk` and `my_in2` is hooked up to the `clk` and `my_in2` signals.

Interfaces

Interfaces provide a way to specify communication between blocks. An interface is a group of nets and variables that are grouped together for the purpose of making connections between modules easier to write.

The syntax for a basic interface is:

```
interface interface_name;
parameters and ports;
items;
endinterface : interface_name
```

The `interface_name` at the end is optional but makes the code easier to read. For an example, see the following code:

```
module bottom1 (
input clk,
input [9:0] d1,d2,
input s1,
input [9:0] result,
output logic sel,
output logic [9:0] data1, data2,
output logic equal);

//logic//

endmodule

module bottom2  (
input clk,
input sel,
input [9:0] data1, data2,
output logic [9:0] result);

//logic//

endmodule

module top (
input clk,
input s1,
input [9:0] d1, d2,
output equal);

logic [9:0] data1, data2, result;
logic sel;
```

```
bottom1 u0 (clk, d1, d2, s1, result, sel, data1, data2, equal);
bottom2 u1 (clk, sel, data1, data2, result);
endmodule
```

The previous code snippet instantiates two lower-level modules with some signals that are common to both. These common signals can all be specified with an interface:

```
interface my_int
logic sel;
logic [9:0] data1, data2, result;
endinterface : my_int
```

Then in the two bottom-level modules, you can change to:

```
module bottom1 (
my_int int1,
input clk,
input [9:0] d1, d2,
input s1,
output logic equal);
```

and:

```
module bottom2 (
my_int int1,
input clk);
```

Inside the modules, you can also change how you access `sel`, `data1`, `data2`, and `result`. This is because, according to the module, there are no ports of these names. Instead, there is a port called `my_int`. This requires the following change:

```
if (sel)
result <= data1;
```

to:

```
if (int1.sel)
int1.result <= int1.data1;
```

Finally, in the top level, the interface needs to be instantiated, and the instances reference the interface:

```
module top(
input clk,
input s1,
input [9:0] d1, d2,
output equal);
my_int int3();  //instantiation
```

```
bottom1 u0 (int3, clk, d1, d2, s1, equal);
bottom2 u1 (int3, clk);
endmodule
```

Modports

In the previous example, the signals inside the interface are no longer expressed as inputs or outputs. Before the interface was added, the port `sel` was an output for `bottom1` and an input for `bottom2`.

After the interface is added, that is no longer clear. In fact, the Vivado synthesis engine does not issue a warning that these are now considered bidirectional ports, and in the netlist generated with hierarchy, these are defined as `inouts`. This is not an issue with the generated logic, but it can be confusing.

To specify the direction, use the `modport` keyword, as shown in the following code snippet:

```
interface my_int;
        logic sel;
        logic [9:0] data1, data2, result;

        modport b1 (input result, output sel, data1, data2);
        modport b2 (input sel, data1, data2, output result);
endinterface : my_int
```

Then in the bottom modules, use when declared:

```
module bottom1 (
  my_int.b1 int1,
```

This correctly associates the inputs and outputs.

Miscellaneous Interface Features

In addition to signals, there can also be tasks and functions inside the interface. This lets you create tasks specific to that interface.

Interfaces can be parameterized. In the example above, `data1` and `data2` were both 10-bit vectors, but those can be modified to be any size depending on a parameter that is set.

Packages

Packages provide an additional way to share different constructs. They have similar behavior to VHDL packages. Packages can contain functions, tasks, types, and enums. The syntax for a package is:

```
package package_name;
items
endpackage : package_name
```

The final `package_name` is not required, but it makes code easier to read.

Packages are then referenced in other modules by the import command. Following is the syntax:

```
import package_name::item or *;
```

The `import` command must include items from the package to import or must specify the whole package.

Appendix C

Additional Resources

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx Support website at: www.xilinx.com/support.

For a glossary of technical terms used in Xilinx documentation, see: www.xilinx.com/company/terms.htm.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

Vivado Documentation

Vivado Design Suite 2012.2 Documentation
www.xilinx.com/support/documentation/dt_vivado_vivado2012-2.htm

1. *Xilinx Design Tools: Release Notes Guide (UG631)*
2. *Xilinx Design Tools: Installation and Licensing Guide (UG798)*
3. *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)*
4. *Vivado Tcl Command Reference Guide (UG835)*
5. *Vivado Design Suite User Guide: Using the Tcl Scripting Capabilities (UG894)*
6. *Vivado Design Suite User Guide: Implementation (UG904)*
7. *Vivado Design Suite Migration Methodology Guide (UG912)*
8. *Vivado Design Suite User Guide: Design Flows Overview (UG892)*
9. *PlanAhead User Guide (UG632)*
10. *XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices (UG687)*