

Vivado Design Suite User Guide

Implementation

UG904 (v2013.1) March 20, 2013



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 2012-2013 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
03/20/2013	2013.1	<p>Added information on Using the -no_timing_driven Option and Using the -verbose Option.</p> <p>Substantially expanded section on Implementation Strategies.</p> <p>Updated Manual Routing section.</p> <p>Added new section Locking Cell Inputs on LUT Loads.</p> <p>Added new section Using Directives.</p> <p>Renamed former section Physical Synthesis to Physical Optimization to make it align better with command name.</p> <p>Added substantial new information on Physical Optimization.</p> <p>Added new section Physical Optimization Constraints.</p> <p>Provided greater detail on phys_opt_design options.</p> <p>Updated information on Implementation Commands.</p> <p>Substantially updated and modified Appendix A, Using Remote Hosts.</p> <p>Changed "instances" to "cells" and "attributes" to "properties" to be consistent with Tcl command terminology (except when referring to HDL keywords).</p> <p>Removed former Chapter 2, Defining Relatively Placed Macros. This included a new section on XDC macros. The entire chapter will be relocated to <i>Vivado Design Suite User Guide: Using Constraints (UG903)</i>.</p> <p>Minor language and formatting edits throughout.</p> <p>Updated various figures.</p> <p>Edited coding examples for accuracy.</p>

Table of Contents

Revision History	2
Chapter 1: Vivado Implementation Process	
About the Vivado Implementation Process	5
Getting to Implementation	8
Configuring, Implementing, and Verifying IP	12
Guiding Implementation With Design Constraints	13
Saving and Restoring Snapshots of a Design with Design Checkpoints	16
Running Implementation in Non-Project Mode	17
Running Implementation in Project Mode.	20
Customizing Implementation Strategies	32
Launching Implementation Runs	39
Moving Processes into the Background	41
Running Implementation in Steps	42
Monitoring the Implementation Run	43
Saving Placer and Router Runtime with Incremental Compile.	47
Moving Forward After Implementation	55
Viewing Messages	58
Viewing Implementation Reports.	60
Chapter 2: Implementation Commands	
About Implementation Commands	65
Implementation Sub-Processes	65
Opening the Synthesized Design.	66
Logic Optimization	70
Power Optimization.	73
Placement.	75
Physical Optimization	80
Routing	84

Chapter 3: Modifying Routing and Logic

Introduction to Modifying Routing and Logic	90
Modifying Routing	90
Modifying Logic	101

Appendix A: Using Remote Hosts

Launching Runs on Remote Linux Hosts	104
Setting Up SSH Key Agent Forward	108

Appendix B: ISE Command Map

Tcl Commands and Options	109
--------------------------------	-----

Appendix C: Additional Resources

Xilinx Resources	112
Solution Centers	112
References	112

Vivado Implementation Process

About the Vivado Implementation Process

The Xilinx® Vivado™ Design Suite enables implementation of Xilinx 7 series FPGA designs from a variety of design sources, including:

- RTL designs
- Netlist designs
- IP centric design flows

See [Figure 1-1, Vivado Design Suite High-Level Design Flow, page 6](#).

Vivado implementation includes all steps necessary to place and route the netlist onto the FPGA device resources, while meeting the design's logical, physical, and timing constraints.

For more information about the design flows supported by the Vivado tools, see the *Vivado Design Suite User Guide: Design Flows Overview (UG892)* [\[Ref 1\]](#).

Vivado Implementation Supports SDC and XDC Constraints

The Vivado Design Suite implementation is a timing-driven flow. It supports industry standard Synopsys Design Constraints (SDC) commands to specify design requirements and restrictions, as well as additional commands in the Xilinx Design Constraints format (XDC).

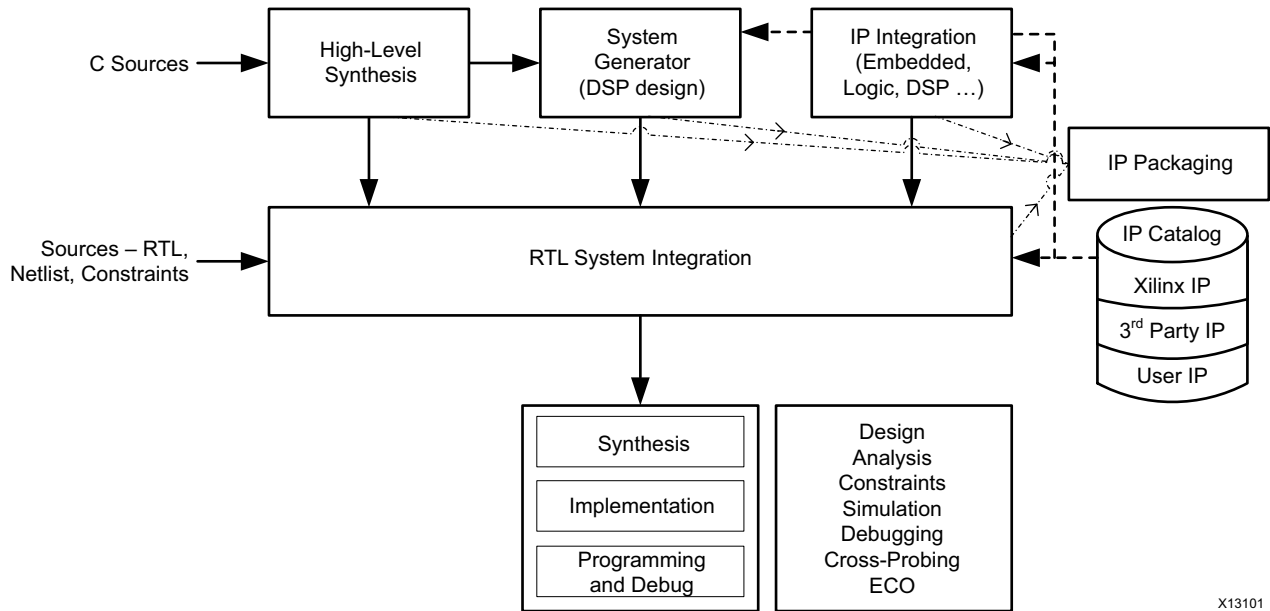


Figure 1-1: Vivado Design Suite High-Level Design Flow

Vivado Implementation Sub-Processes

The Vivado Design Suite implementation process includes logical and physical transformations of the design. The implementation process consists of the following sub-processes:

- **Opt Design**

Optimizes the logical design to make it easier to fit onto the target Xilinx device.

- **Power Opt Design**

Optimizes design elements to reduce the power demands of the target Xilinx device.

Note: This step is optional.

- **Place Design**

Places the design onto the target Xilinx device.

- **Phys Opt Design**

Optimizes design timing by replicating drivers of high-fanout nets to distribute the loads.

Note: This step is optional.

- **Route Design**

Routes the design onto the target Xilinx device.

- **Write Bitstream**

Generates a bitstream for Xilinx device configuration.

Flow Navigator Assembles, Implements, and Validates Your Design

The complete design flow is integrated in the Vivado Integrated Design Environment (IDE). The Vivado IDE includes a standardized interface called the Flow Navigator.

The Flow Navigator assembles, implements, and validates the design and IP of the FPGA design. It features a push-button interface to the entire implementation process to simplify the design flow.

See [Figure 1-2, Flow Navigator - Implementation Section](#).

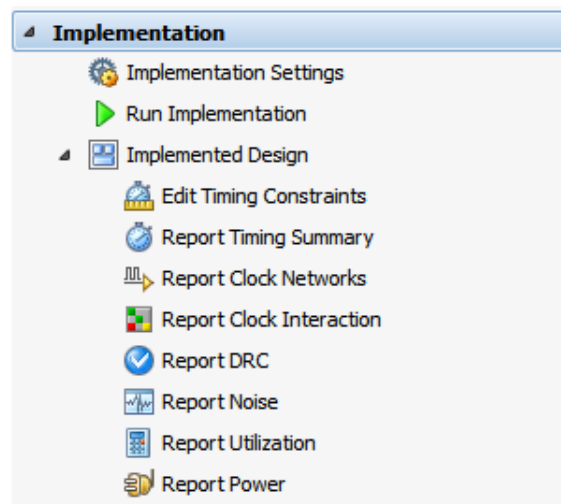


Figure 1-2: Flow Navigator - Implementation Section



IMPORTANT: This guide does not give a detailed explanation of the Vivado IDE, except as it applies to implementation. For more information about the Vivado IDE as it relates to the entire FPGA design flow, see the *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)* [Ref 2].

Tcl API Supports Scripting

The Vivado Design Suite includes a Tool Command Language (Tcl) Application Programming Interface (API). The Tcl API supports scripting for all design flows, allowing you to customize the design flow to meet your specific requirements.

Note: For more information about Tcl commands, see the *Vivado Design Suite Tcl Command Reference Guide (UG835)* [Ref 12], or type `<command> -help`.

Getting to Implementation

The Vivado Design Suite includes a variety of design flows, and supports an array of design sources. In order to generate a bitstream that can be downloaded onto an FPGA device, the design must pass through implementation.

Implementation is a series of steps that takes the logical netlist and maps it into the physical array of the target Xilinx device. These steps include:

- Logic optimization
- Placement of logic cells
- Routing of connections between cells

Working in Project Mode and Non-Project Mode

The Vivado Design Suite lets you create a project file (.xpr) and directory structure that allows you to:

- Manage the design source files.
- Store the results of the synthesis and implementation runs.
- Track the project status through the design flow.



TIP: *The Vivado tools also let you work strictly in memory, without the need for a project file and local directory.*

Working in Project Mode

In Project Mode, a directory structure is created on disk to help you manage:

- Design sources
- Run results
- Project status

The automated management of the design data, process, and status requires a project infrastructure that is stored in the Vivado project file (.xpr).

In Project Mode, the Vivado tool automatically writes checkpoint files into the local project directory at key points in the design flow.

Working in Non-Project Mode

Working without a project file in the compilation style flow is called Non-Project Mode. Non-Project Mode allows you to work with the design in memory. Source files and design constraints are read into memory from their current locations. The in-memory design is stepped through the design flow without being written to intermediate files.

In Non-Project Mode, you must run each design step individually, and set design parameters and implementation options using Tcl commands.

Non-Project Mode allows you to apply design changes and proceed through the design flow without needing to save changes and rerun steps. You can run reports and save design checkpoints (.dcp) at any stage of the design flow.



IMPORTANT: *In Non-Project Mode, when you exit the Vivado design tools, the in-memory design is lost. For this reason, Xilinx recommends that you write design checkpoints after major steps such as synthesis, placement, and routing.*

You can save design checkpoints in both Project Mode and Non-Project Mode. You can only read design checkpoints in Non-Project Mode.

Similarities and Differences Between Project Mode and Non-Project Mode

Vivado implementation can be run in either Project Mode or Non-Project Mode. The Vivado IDE and Tcl API can be used in both Project Mode and Non-Project Mode.

There are also many differences between Project Mode and Non-Project Mode. Features not available in Non-Project Mode include:

- Flow Navigator
- Design status indicators
- IP catalog
- Implementation runs and run strategies
- Design Runs window
- Messages window
- Reports window

Note: This list illustrates features that are not supported in Non-Project Mode. It is not exhaustive.

You must implement the non-project based design by running the individual Tcl commands:

- **opt_design**
- **place_design**
- **route_design**

You can run implementation steps interactively in the Tcl Console or the Vivado IDE, or by using a custom Tcl script. You can customize the design flow as needed to include reporting commands and additional optimizations. For more information, see [Running Implementation in Non-Project Mode](#).

The details of running implementation in Project Mode and Non-Project Mode are described in this guide.

For more information on running the Vivado Design Suite using either Project Mode or Non-Project Mode, see:

- *Vivado Design Suite User Guide: Design Flows Overview (UG892)* [Ref 1]
- *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)* [Ref 2]

RTL and Synthesized Design

The Vivado Design Suite allows you to manage the entire FPGA design process, including RTL development, IP customization, synthesis, and implementation through to programming and validating the device.

Adding Objects to Your Project

You can add the following objects to your project:

- HDL source files from Verilog, SystemVerilog, and VHDL
- Previously defined and configured Xilinx IP cores
- Digital signal processing (DSP) modules from System Generator.
- C-based DSP modules from Vivado High-level Synthesis (HLS)
- Embedded processor modules from Xilinx Platform Studio (XPS)

Importing Previously Synthesized Netlists

Vivado Design Suite supports netlist-driven design by importing previously synthesized netlists from Xilinx or third-party tools. The netlist input formats include:

- Structural Verilog
- SystemVerilog
- EDIF
- Xilinx NGC

For more information on the source files and project types supported by the Vivado Design Suite, see the *Vivado Design Suite User Guide: System-Level Design Entry (UG895)* [Ref 4].

Starting From RTL Sources

At a minimum, Vivado implementation requires a synthesized netlist. A design can start from a synthesized netlist, or from RTL source files.



IMPORTANT: *If you start from RTL sources, you must first run either Vivado synthesis or XST before implementation can begin. The Vivado IDE manages this automatically if you attempt to run implementation on an un-synthesized design. The tool allows you to run synthesis first.*

For information on running Vivado synthesis, see the *Vivado Design Suite User Guide: Synthesis (UG901)* [Ref 6].

Creating and Opening the Synthesized Design in Non-Project Mode

In Non-Project Mode, you must run the Tcl command `synth_design` to create and open the synthesized design. You can also run the Tcl command `link_design` to open a synthesized netlist in any supported input format.

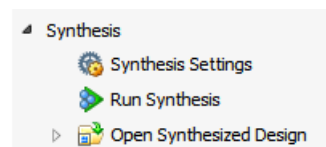
For more information, see [Opening the Synthesized Design](#) in [Chapter 2, Implementation Commands](#).

Loading the Design Netlist in Project Mode Before Implementation

In Project Mode, after synthesis of an RTL design, or with a netlist-based project open, you can load the design netlist for analysis before implementation.

To open a synthesized design, do one of the following:

- From the main menu, run **Flow > Open Synthesized Design**.
- In the Flow Navigator, run **Synthesis > Open Synthesized Design**.
- In the Design Runs window, select the synthesis run and select **Open Synthesized Design** from the context menu.



Configuring, Implementing, and Verifying IP

The Vivado IP catalog allows you to configure, implement, and verify IP. The IP can be configured and verified as a standalone module, or within the context of a larger system level design.

IP Catalog Contents

The IP catalog displays all available Xilinx LogicCORE™ IP and user-defined IP or third party IP that has been added to the IP catalog.

The catalog includes data related to:

- IP type
- Version
- Datasheet
- License information

Adding an IP Core to an RTL Design

To add an IP core to an RTL design, define the instantiation template into the system-level design.

IP is created as RTL sources, not netlists. Running synthesis and implementation implements the IP along with the rest of the design.

You can also synthesize the IP as a standalone module, and add the netlist to a netlist design.

Table 1-1: Supported IP Netlist Formats

Xilinx	Verilog	EDIF
<ul style="list-style-type: none">• .xco• .xci• .ngc	.v	.edf

For more information on how Vivado tools support IP centric design, see the *Vivado Design Suite User Guide: Designing with IP (UG896)* [Ref 5].

Guiding Implementation With Design Constraints

Xilinx highly recommends that you include design constraints to guide implementation. There are two types of design constraints, physical constraints and timing constraints.

This section includes the following:

- [What Physical Constraints Define](#)
- [What Timing Constraints Define](#)
- [UCF Format Not Supported](#)
- [Constraint Sets Apply Lists of Constraint Files to Your Design](#)
- [Adding Constraints as Attribute Statements](#)

What Physical Constraints Define

Physical constraints define:

- Pin placement
- Absolute or relative placement of cells, including:
 - BRAM
 - DSP
 - LUT
 - Flip flops
- Device configuration settings

What Timing Constraints Define

Timing constraints define the frequency requirements for the design, and are written in industry standard SDC.

Without timing constraints, the Vivado Design Suite optimizes the design solely for wire length and routing congestion, and makes no effort to assess or improve design performance.

UCF Format Not Supported



IMPORTANT: *The Vivado Design Suite does not support the UCF format.*

For information on migrating UCF constraints to XDC commands, see the *Vivado Design Suite Migration Methodology Guide (UG911)* [Ref 13].

Constraint Sets Apply Lists of Constraint Files to Your Design

A constraint set is a list of constraint files that can be applied to your design. The set contains design constraints captured in XDC files.

Allowed Constraint Set Structures

The following constraint set structures are allowed:

- Multiple constraint files within a constraint set
- Constraint sets with separate physical and timing constraint files
- A master constraint file
- A new constraint file that accepts constraint changes
- Multiple constraint sets



TIP: *Separate constraints by function into different constraint files to (a) make your constraint strategy clearer, and (b) to facilitate targeting timing and implementation changes.*

Multiple Constraint Sets Are Allowed

You can have multiple constraint sets for a project. Multiple constraint sets allow you to use different implementation runs to test different approaches.

For example, you can have one constraint set for synthesis, and a second constraint set for implementation. Having two constraint sets allows you to experiment by applying different constraints during synthesis, simulation, and implementation.

Organizing design constraints into multiple constraint sets can help you:

- Target various Xilinx FPGA devices for the same project. Different physical and timing constraints may be needed for different target parts.
- Perform *what-if* design exploration. Use constraint sets to explore various scenarios for floorplanning and over-constraining the design.
- Manage constraint changes. Override master constraints with local changes in a separate constraint file.



TIP: To validate the timing constraints, run `report_timing_summary` on the synthesized design. Fix problematic constraints before implementation!

For more information on defining and working with constraints that affect placement and routing, see the *Vivado Design Suite User Guide: Using Constraints (UG903)* [Ref 7].

Adding Constraints as Attribute Statements

Constraints can be added to HDL sources as attribute statements. Attributes can be added to both Verilog and VHDL sources to pass through to Vivado synthesis or Vivado implementation. In some cases, constraints are available only as RTL attributes, and are not available in XDC.

In this case, the constraint must be specified as an attribute in the HDL source file. For example, Relatively Placed Macros (RPMs) must be defined as properties. An RPM is a list of logic elements (such as FF, LUT, DSP, and RAM) grouped into a set.

You can define sets of design elements using U Set (U_SET) or HU Set (HU_SET) constraints, and place these objects in relation to the other elements of the set using Relative Location Constraints (RLOC).

For more information about Relative Location Constraints, see the *Vivado Design Suite User Guide: Using Constraints (UG903)* [Ref 7].

The U_SET, HU_SET, and RLOC constraints are not supported in XDC by Tcl commands, and they must be defined as attributes in the HDL source files.

For more information on constraints that are *not* supported in XDC, see the *Vivado Design Suite Migration Methodology Guide (UG911)* [Ref 13].

Saving and Restoring Snapshots of a Design with Design Checkpoints

The Vivado Design Suite uses a physical design database to store placement and routing information. Design checkpoint files (.dcp) allow you to save and restore this physical database at key points in the design flow. Checkpoints are a snapshot of a design at a specific point in the flow.

This design checkpoint file includes the following:

- Current netlist, including any optimizations made during implementation
- Design constraints
- Implementation results

Checkpoint designs can be run through the remainder of the design flow using Tcl commands. They cannot be modified with new design sources.

Writing Checkpoint Files

Run **File > Write Checkpoint** to capture a snapshot of the design database at any point in the flow. This creates a file with a dcp file extension.

The related Tcl command is **write_checkpoint**.

Reading Checkpoint Files

Run **File > Open Checkpoint** to open the checkpoint in the Vivado Design Suite.

The design checkpoint is opened as a separate project. It can not be read into an existing project.

The related Tcl command is **read_checkpoint**.

Running Implementation in Non-Project Mode

To implement the synthesized design or netlist onto the targeted Xilinx FPGA device, you must run the netlist and the design constraints through a series of steps:

- Optimization
- Placement
- Routing

These steps are collectively known as *implementation*.

In Non-Project Mode, you must run implementation using a series of Tcl commands, or a Tcl script that defines the design flow. The Tcl commands can be entered into the Tcl Console from the Vivado IDE, or from the Tcl prompt in the Vivado Design Suite Tcl shell.

Non-Project Mode Example Script

The following script is an example of running implementation in Non-Project Mode.

```
# Step 1: Read in top-level EDIF netlist from synthesis tool
read_edif c:/top.edf
# Read in lower level IP core netlists
read_edif c:/core1.edf
read_edif c:/core2.edf

# Step 2: Specify target device and link the netlists
# Merge lower level cores with top level into single design
link_design -part xc7k325tfbg900-1 -top top.edf

# Step 3: Read XDC constraints to specify timing requirements
read_xdc c:/top_timing.xdc
# Read XDC constraints that specify physical constraints such as pin locations
read_xdc c:/top_physical.xdc

# Step 4: Optimize the design with default settings
opt_design

# Step 5: Place the design
place_design

# Step 6: Route the design
route_design

# Step 7: Run Timing Summary Report to see timing results
report_timing_summary -file post_route_timing.rpt
# Run Utilization Report for device resource utilization
report_utilization -file post_route_utilization.rpt

# Step 8: Write checkpoint to capture the design database;
# The checkpoint can be used for design analysis in Vivado IDE or TCL API
write_checkpoint post_route.dcp
```

Key Steps in Non-Project Mode Example Script

The key steps in the [Non-Project Mode Example Script](#) above are:

- [Step 1: Read Design Source Files](#)
- [Step 2: Build the In-Memory Design](#)
- [Step 3: Read Design Constraints](#)
- [Step 4: Perform Logic Optimization](#)
- [Step 5: Place the Netlist Elements](#)
- [Step 6: Route the Design](#)
- [Step 7: Run Required Reports](#)
- [Step 8: Save the Design Checkpoint](#)

Step 1: Read Design Source Files

In the [Non-Project Mode Example Script](#) above, the design sources are EDIF netlist files. Non-Project Mode also supports an RTL design flow, which allows you to read source files and run synthesis before implementation.

The `read_*` Tcl commands are designed for use with Non-Project Mode. This allows a file on the disk to be read by the Vivado Design Suite to build the in-memory design, without copying the file or creating a dependency on the file.

This approach makes Non-Project Mode extremely flexible with regard to design.



IMPORTANT: *You must monitor any changes to the source design files, and update the design as needed.*

Step 2: Build the In-Memory Design

In the [Non-Project Mode Example Script](#) above, the Vivado tools build an in-memory view of the design using `link_design`. The `link_design` command combines (a) the netlist based source files read into the tool with (b) the Xilinx part information, to create a design database in memory.

All actions taken in Non-Project Mode are directed at the in-memory database within the Vivado tools.

The in-memory design resides in the Vivado tool, whether running in batch mode, Tcl shell mode for interactive Tcl commands, or in the Vivado IDE for interaction with the design data in a graphical form.

Step 3: Read Design Constraints

The Vivado Design Suite uses design constraints to define requirements for both the physical and timing characteristics of the design.

For more information, see [Guiding Implementation With Design Constraints, page 13](#).

The `read_xdc` command reads an XDC constraint file, then applies it to the in-memory design.



TIP: While Project Mode supports the definition of constraint sets, containing multiple constraint files for different purposes, Non-Project Mode uses multiple `read_xdc` commands to achieve the same effect.

Step 4: Perform Logic Optimization

The [Non-Project Mode Example Script](#) above performs logic optimization in preparation for placement and routing. Optimization simplifies the logic design before committing to physical resources on the target part.

The Vivado netlist optimizer includes many different types of optimizations to meet varying design requirements.

For more information, see [Logic Optimization, page 70](#).

Step 5: Place the Netlist Elements

The [Non-Project Mode Example Script](#) above performs a general placement of the design. Placement can also be accomplished in stages, according to the design hierarchy, or the complexity of the placement challenge.

For more information, see [Placement, page 75](#).

Step 6: Route the Design

The `route_design` command completes the required routing for the design. The Vivado router performs timing-driven routing for all design types. The router allows a great deal of control for re-entrant routing to complete challenging designs.

For more information, see [Routing, page 84](#).

Step 7: Run Required Reports

The [Non-Project Mode Example Script](#) above generates two of the many reports available from the Vivado Design Suite. In Non-Project Mode, you must use the appropriate Tcl command to specify each report you want to create.

You can output reports to files for later review, or you can send the reports directly to the Vivado IDE to review now.

For more information, see [Viewing Implementation Reports, page 60](#).

Step 8: Save the Design Checkpoint

The [Non-Project Mode Example Script](#) above saves the in-memory design into a design checkpoint file. The saved in-memory design includes its:

- Optimized netlist
- Physical and timing related constraints
- Xilinx part data
- Placement and routing information

In Non-Project Mode, the design checkpoint file saves the design and allows it to be reloaded for further analysis and modification.

For more information, see [Saving and Restoring Snapshots of a Design with Design Checkpoints, page 16](#).

Running Implementation in Project Mode

In Project Mode, the Vivado IDE allows you to:

- Define implementation runs that are configured to use specific synthesis results and design constraints.
- Run multiple strategies on a single design.
- Customize implementation strategies to meet specific design requirements.
- Save customized implementation strategies to use in other designs.



IMPORTANT: *Non-Project Mode does not support predefined implementation runs and strategies. Non-project based designs must be manually moved through each step of the implementation process using Tcl commands. For more information, see [Running Implementation in Non-Project Mode, page 17](#).*

Creating Implementation Runs

You can create and launch new implementation runs to explore design alternatives and find the best results. You can queue and launch the runs serially, or in parallel using multiple local CPUs.

On Linux systems, you can launch runs on remote servers. For more information, see [Appendix A, Using Remote Hosts](#).

Defining Implementation Runs

To define an implementation run:

- Do one of the following:
 - From the main menu, select **Flow > Create Runs**.
 - In the Flow Navigator, select **Create Implementation Runs** from the Implementation popup menu.
 - In the Design Runs window, select **Create Runs** from the popup menu.

The Create New Runs wizard opens. The first page summarizes the command.

- Click **Next**.

Note: If you used **Flow > Create Runs**, select **Implementation** on the first page of the Create New Runs wizard.

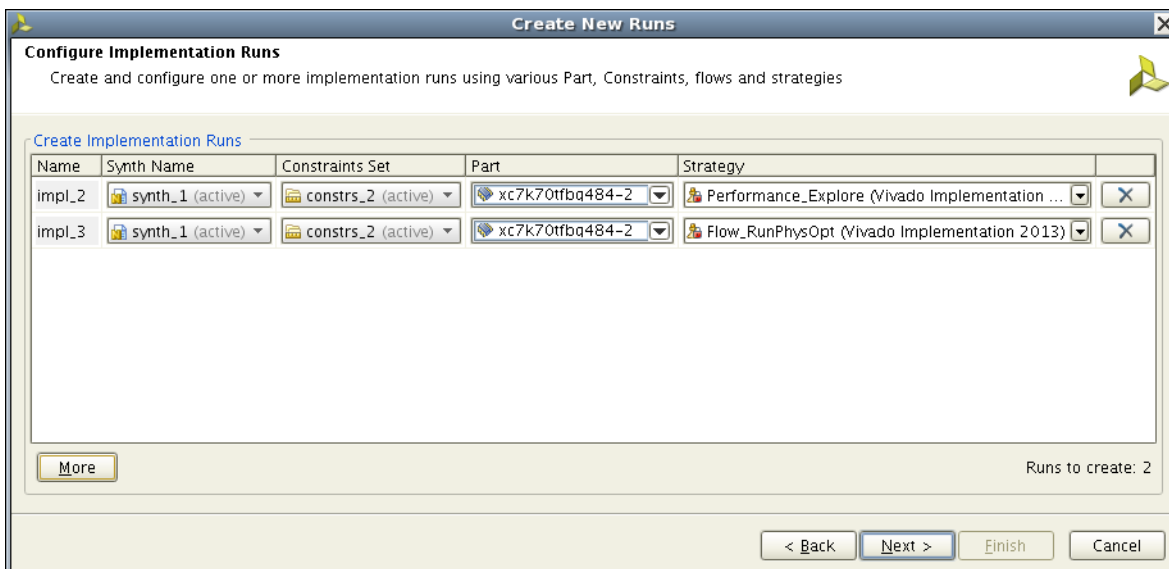


Figure 1-3: Configure Implementation Runs

- Enter a **Name** for the run in the Configure Implementation Runs dialog box, or accept the default name.

4. Select a **Synth Name** to choose the synthesis run that will be used to generate (or that has already generated) the synthesized netlist to be implemented.
 - Alternatively, you can select a synthesized netlist that was imported into the project from a third party synthesis tool.
 - For more information, see the *Vivado Design Suite User Guide: Synthesis (UG901)* [Ref 6].
 - The default is the currently active synthesis run in the Design Runs window. For more information, see [Using the Design Runs Window, page 26](#).
5. Select a **Constraints Set**.
 - Select a **Constraints Set** to specify the constraint set to apply during implementation. The optimization, placement, and routing are largely directed by the physical and timing constraints in the specified constraint set.
 - For more information on constraint sets, see the *Vivado Design Suite User Guide: Using Constraints (UG903)* [Ref 7].
6. Select a target **Part**.
 - The default values for Constraints Set and Part are defined by the Project Settings when the **Create New Runs** command is executed.
 - For more information on the Project Settings, see the *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)* [Ref 2].
 - To create runs with different constraint sets or target parts, use the **Create New Runs** command. To change these values in the Run Properties window, select the run in the Design Runs window.
 - For more information, see [Changing Implementation Run Settings, page 28](#).
7. Select a **Strategy**.
 - Strategies are a defined set of Vivado implementation feature options controlling the implementation results. Vivado Design Suite includes a set of pre-defined strategies. You can also create your own implementation strategies.
 - For more information see [Defining Strategies, page 34](#).
 - Select from among the strategies shown in the following table.

The strategies are broken into categories according to their purposes, with the category name as a prefix. The categories are shown in [Table 1-2, Categories](#).

The Performance strategies aim to improve design performance at the expense of runtime. The **Performance_Explore** strategy is a good first choice, because it covers all types of designs.



IMPORTANT: *Strategies containing the terms SLL or SLR are for use with SSI devices only.*

Table 1-2: Categories

Category	Purpose
Performance	Improve design performance.
Area	Reduce LUT count.
Power	Add full power optimization.
Flow	Modify flow steps.
Congestion	Reduce congestion and related problems.

Table 1-3: Implementation Strategies

Implementation Strategy Name	Description
Vivado Implementation Defaults	Balances runtime with trying to achieve timing closure.
Performance_Explore	Uses multiple algorithms for optimization, placement, and routing to get potentially better results.
Performance_RefinePlacement	Increase placer effort in the post-placement optimization phase, and disable timing relaxation in the router.
Performance_WLBlockPlacement	Ignore timing constraints for placing Block RAM and DSPs, use wirelength instead.
Performance_WLBlockPlacementFanoutOpt	Ignore timing constraints for placing Block RAM and DSPs, use wirelength instead, and perform aggressive replication of high fanout drivers.
Performance_LateBlockPlacement	Use approximate Block RAM and DSP placement until late placement phases. May result in better overall placement.
Performance_NetDelay_high	To compensate for optimistic delay estimation, add extra delay cost to long distance and high fanout connections. (high setting, most pessimistic)
Performance_NetDelay_medium	To compensate for optimistic delay estimation, add extra delay cost to long distance and high fanout connections. (medium setting,)
Performance_NetDelay_low	To compensate for optimistic delay estimation, add extra delay cost to long distance and high fanout connections. low setting, least pessimistic)
Performance_ExploreSLs	Explores SLR reassignments to potentially improve overall timing slack.
Area_Explore	Uses multiple optimization algorithms to get potentially fewer LUTs.
Power_DefaultOpt	Adds power optimization (power_opt_design) to reduce power consumption.
Flow_RunPhysOpt	Similar to the Implementation Run Defaults, but enables the physical optimization step (phys_opt_design).

Table 1-3: Implementation Strategies (Cont'd)

Implementation Strategy Name	Description
Flow_RuntimeOptimized	Each implementation step trades design performance for better runtime. Physical optimization (phys_opt_design) is disabled.
Flow_Quick	Only placement and routing are run, with all optimization and timing-driven behavior disabled. Useful for utilization estimation.
Congestion_SpreadLogic_high	Spread logic throughout the device to avoid creating congested regions. (high setting: highest degree of spreading)
Congestion_SpreadLogic_medium	Spread logic throughout the device to avoid creating congested regions. (medium setting)
Congestion_SpreadLogic_low	Spread logic throughout the device to avoid creating congested regions. (low setting: lowest degree of spreading)
Congestion_SpreadLogicSLLs	Allocate SLLs such that logic can be spread throughout all SLRs to avoid creating congested regions inside SLRs.
Congestion_BalanceSLLs	Allocate SLLs such that no two SLRs require a disproportionately large number of SLLs, thereby reducing congestion in those SLRs.
Congestion_BalanceSLRs	Partition such that each SLR has similar area, to avoid creating congestion within an SLR.
Congestion_CompressSLRs	Partition with higher SLR utilization, to reduce number of overall SLLs.



TIP: Before launching a run, you can change the settings for each step in the implementation process, overriding the default settings for the selected strategy. You can also save those new settings as a new strategy. For more information, see [Changing Implementation Run Settings, page 28](#).

8. Click **More** to define additional runs. Specify names and strategies for the added runs. See [Figure 1-3, Configure Implementation Runs](#).
9. Click **Next**.

The Launch Options dialog box opens. See [Figure 1-4, Implementation Launch Options](#).

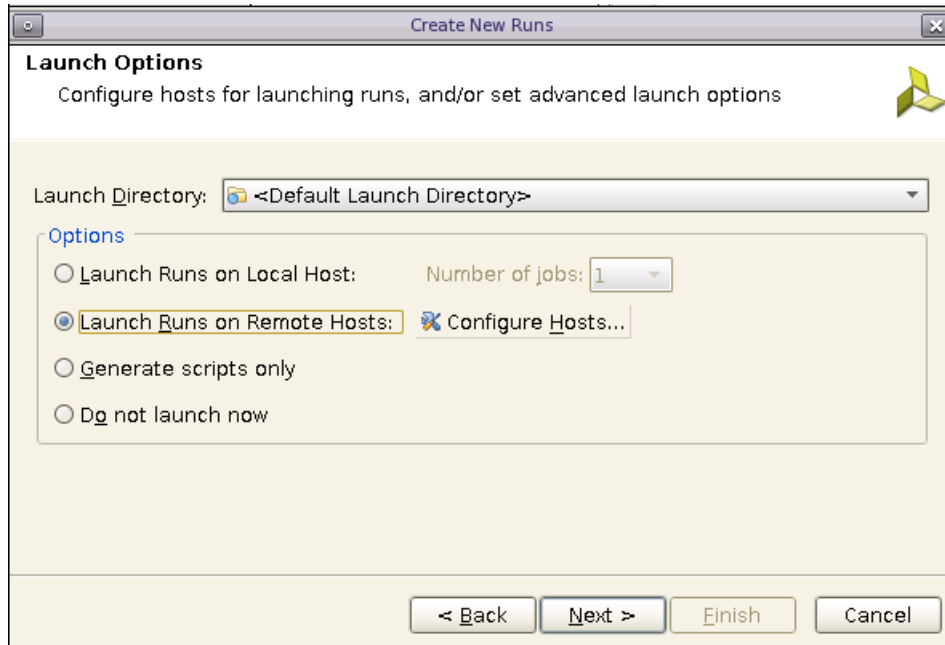


Figure 1-4: Implementation Launch Options

10. Specify the Launch Directory

- Creates and stores the implementation run data.
- The default directory is located in the local project directory structure. Files for implementation runs are stored by default at:

`<project_name>/<project_name>.runs/<run_name>`



TIP: Defining a directory location outside the project directory structure makes the project non-portable because absolute paths are written into the project files.

11. Specify the Launch Options.

- **Launch Runs on Local Host**
Launches the run on the local machine.
- **Number of Jobs**
Defines the number of local processors to use when launching multiple runs simultaneously.
- **Launch Runs on Remote Hosts** (Linux only)
 - Uses remote hosts to launch one or more jobs.
 - For more information, see [Appendix A, Using Remote Hosts](#).

- **Configure Hosts**

Configures remote hosts.

- **Generate scripts only**

Exports and creates the run directory and run script, but does not launch the run at this time. The script can be run later outside the Vivado IDE tool.

- **Do not launch now**

Saves the new runs, but does not launch or create run scripts at this time.

12. Click **Next** to review the Create New Runs Summary.

13. Click **Finish** to create the defined runs and execute the specified launch options.

New runs are added to the Design Runs window.

Using the Design Runs Window

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

Opening the Design Runs Window

Select **Window > Design Runs** to open the Design Runs window if it is not already open. See [Figure 1-5, Design Runs Window](#).

Design Runs Window Functionality

- Each implementation run appears indented beneath the synthesis run of which it is a child.
- A synthesis run can have multiple implementation runs. Use the tree widgets in the window to expand and collapse synthesis runs.
- The Design Runs window is a tree table window.

For more information on working with the columns to sort the data in this window, see the *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)* [\[Ref 2\]](#).

Name	Part	Constraints	Strategy	Status
synth_1	xc7k70tfg484-2	constrs_2	Vivado Synthesis Defaults (Vivado Synthesis 20...	synth_design Complete!
impl_1 (active)	xc7k70tfg484-2	constrs_2	Vivado Implementation Defaults (Vivado Im...	route_design Complete!
impl_2	xc7k70tfg484-2	constrs_2	Performance_Explore (Vivado Implementation ...	Not started
impl_3	xc7k70tfg484-2	constrs_2	Flow_RunPhysOpt (Vivado Implementation 2013)	Not started

Figure 1-5: Design Runs Window

Run Status

The Design Runs window reports the run status, including when the run:

- Has not been started.
- Is in progress.
- Is complete.
- Is out-of-date.

Run Timing Results

The Design Runs window reports timing results for implementation runs including WNS, TNS, WHS, THS, TPWS, and the number of failed nets.

Out-of-Date Runs

Runs can become out-of-date when source files, constraints, or project settings are modified. You can reset and delete stale run data in the Design Runs window.

Active Run

All views in the Vivado IDE reference the active run. The Log view, Report view, Status Bar, and Project Summary display information for the active run. The Project Summary window displays only compilation, resource, and summary information for the active run.



TIP: Only one synthesis run and one implementation run can be active in the Vivado IDE at any time

The active run is displayed in **bold** text.

To make a run active:

1. Select the run in the Design Runs window
2. Select **Make Active** from the popup menu.

Changing Implementation Run Settings

Select a run in the Design Runs window to display the current configuration of the run in the Run Properties window. See [Figure 1-6, Run Properties Window](#).

In the Run Properties window you can change:

- The **Name** of the run
- The Xilinx **Part** targeted by the run
- The run **Description**
- The **Constraints** set that both drives the implementation and is the target of new constraints from implementation

For more information on the Run Properties window, see the *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)* [\[Ref 2\]](#).

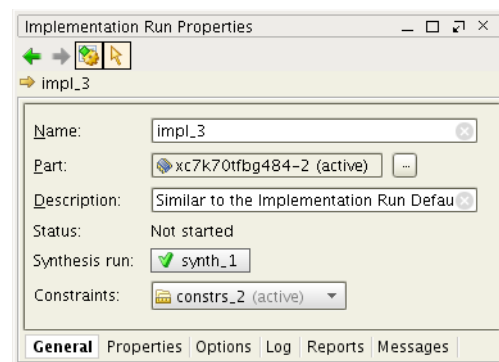


Figure 1-6: Run Properties Window

You can also change the options used by Vivado implementation features.

Specifying Design Run Settings

Specify design run settings in the Design Run Settings dialog box. To open the Design Run Settings dialog box:

1. Select a run in the Design Runs window.
2. Select **Change Run Settings** from the popup menu.

See [Figure 1-7, Design Run Settings](#).



TIP: You can change the settings only for a run that has a **Not Started** status. Use **Reset Run** to return a run to the **Not Started** status. See [Resetting Runs, page 31](#).

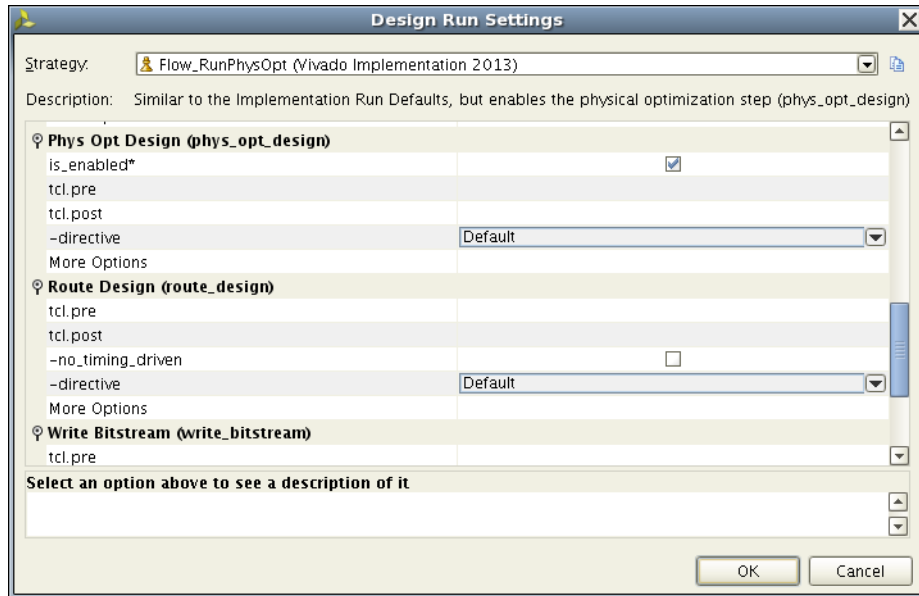


Figure 1-7: Design Run Settings

The Design Run Settings dialog box displays: (1) the implementation strategy currently employed by the run; and (2) the command options associated with that strategy for each step of the implementation process. The command options are:

- [Strategy](#)
- [Description](#)
- [Options](#)

Strategy

Selects the strategy to use for the implementation run. Vivado Design Suite includes a set of pre-defined implementation strategies, or you can create your own.

For more information see [Defining Strategies](#), page 34.

Description

Describes the selected implementation strategy.

Options

When you select a strategy, each step of the Vivado implementation process displays in a table in the lower part of the dialog box:

- **opt_design**
- **power_opt_design**
- **place_design**
- **phys_opt_design**
- **route_design**
- **write_bitstream**

Click a specific command option to view a brief description of the option at the bottom of the Design Run Settings dialog box.

For more information about the various implementation steps, and their available options, see [Chapter 2, Implementation Commands](#).

Modifying Command Options

To modify command options, click the right-side column of a specific option. You can do the following:

- Select options with predefined settings from the pull down menu.
- Select or unselect a check box to enable or disable options.
- Type a value to define options that accept a user-defined value.
- Options accepting a file name and path open a file browser to let you locate and specify the file.
- Insert a custom Tcl script (called a hook script) before and after each step in the implementation process (`tcl.pre` and `tcl.post`).

Inserting a hook script lets you perform specific tasks before or after each implementation step (for example: generate a timing report before and after Place Design to compare timing results).

For more information on defining Tcl hook scripts, see the *Vivado Design Suite User Guide: Using Tcl Scripting (UG894)* [\[Ref 3\]](#).



TIP: Relative paths in the `tcl.pre` and `tcl.post` scripts are relative to the appropriate run directory of the project they are applied to: `<project>/<project.runs>/<run_name>`

Use the DIRECTORY property of the current project or current run to define the relative paths in your Tcl scripts:

```
get_property DIRECTORY [current_project]
get_property DIRECTORY [current_run]
```

Save Strategy As

Select the **Save Strategy As** icon next to the Strategy field to save any changes to the strategy as a new strategy for future use.



CAUTION! If you do not select **Save Strategy As**, changes are saved to the current implementation run, but are not preserved for future use.



Verifying Run Status

The Vivado IDE processes the run, and launches implementation, depending on the status of the run. The status is displayed in the Design Runs window. See [Figure 1-5, Design Runs Window](#).

- If the status of the run is **Not Started**, the run begins immediately.
- If the status of the run is **Error**, the tool resets the run to remove any incomplete run data, then restarts the run.
- If the status of the run is **Complete** (or **Out-of-Date**), the tool prompts you to confirm that the run should be reset before proceeding with the run.

Resetting Runs

To reset a run:

1. Select a run in the Design Runs window.
2. Select **Reset Runs** from the popup menu.

Resetting an implementation run returns it to the first step of implementation (**opt_design**) for the selected run.

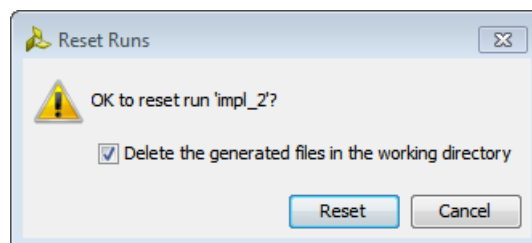


Figure 1-8: Reset Run

The Vivado tool prompts you to confirm the **Reset Runs** command, and optionally delete the generated files from the `run` directory.



TIP: The default setting is to delete the generated files. Disable this check box to preserve the generated run files.

Deleting Runs

To delete runs from the Design Runs window:

1. Select the run.
2. Select **Delete** from the popup menu.

The Vivado tool prompts you to confirm the **Delete Runs** command, and optionally delete the generated files from the `run` directory.



TIP: The default setting is to delete the generated files. Disable this check box to preserve the generated run files.

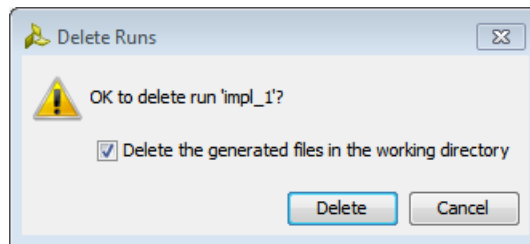


Figure 1-9: Delete Runs

Customizing Implementation Strategies

Implementation Settings define the default options used when you define new implementation runs. Configure these options in the Vivado IDE.

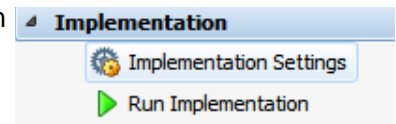
Figure 1-10, [Implementation Settings](#), shows the Implementation Settings of the Project Settings dialog box. To open this dialog box from the Vivado IDE, select **Tools > Project Settings** from the main menu.



TIP: The **Project Settings** command is not available in the Vivado IDE when running in Non-Project Mode. In this case, you can define and preserve implementation strategies as Tcl scripts that can be used in batch mode, or interactively in the Vivado IDE.

Accessing Implementation Settings for the Active Run from Flow Navigator

Implementation Settings for the active implementation run can also be accessed directly from the Flow Navigator.



Implementation Settings contains the following fields :

- **Default Constraint Set**

Select the constraint set to be used by default for the implementation run.

- **Strategy**

Select the strategy to use for the implementation run. Vivado Design Suite includes a set of pre-defined strategies. You can also create your own implementation strategies.

For more information see [Defining Strategies](#).

- **Save Strategy As**

Saves any changes to the strategy as a new strategy for future use.



- **Description**

- Describes the selected implementation strategy.
- The description of user-defined strategies can be changed by entering a new description.
- The description of Vivado tools standard implementation strategies cannot be changed.

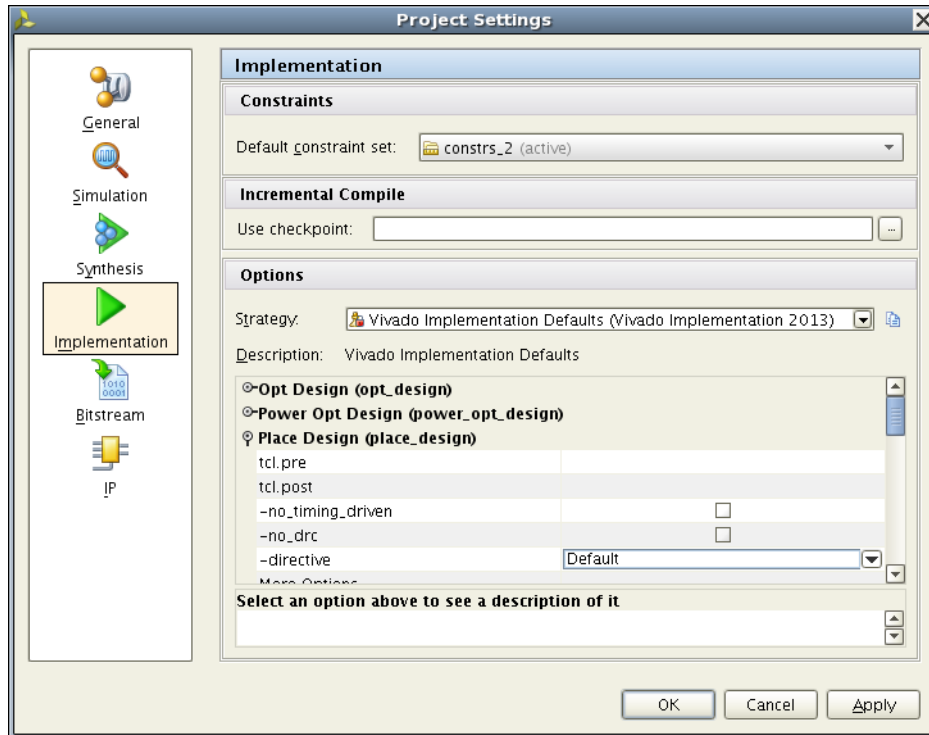


Figure 1-10: Implementation Settings

Defining Strategies

A strategy is a defined approach for resolving the synthesis or implementation challenges of the design.

- Strategies are defined in pre-configured sets of options for the Vivado implementation features.
- Strategies are tool and version specific.
- Each major release of the Vivado Design Suite includes version-specific strategies.

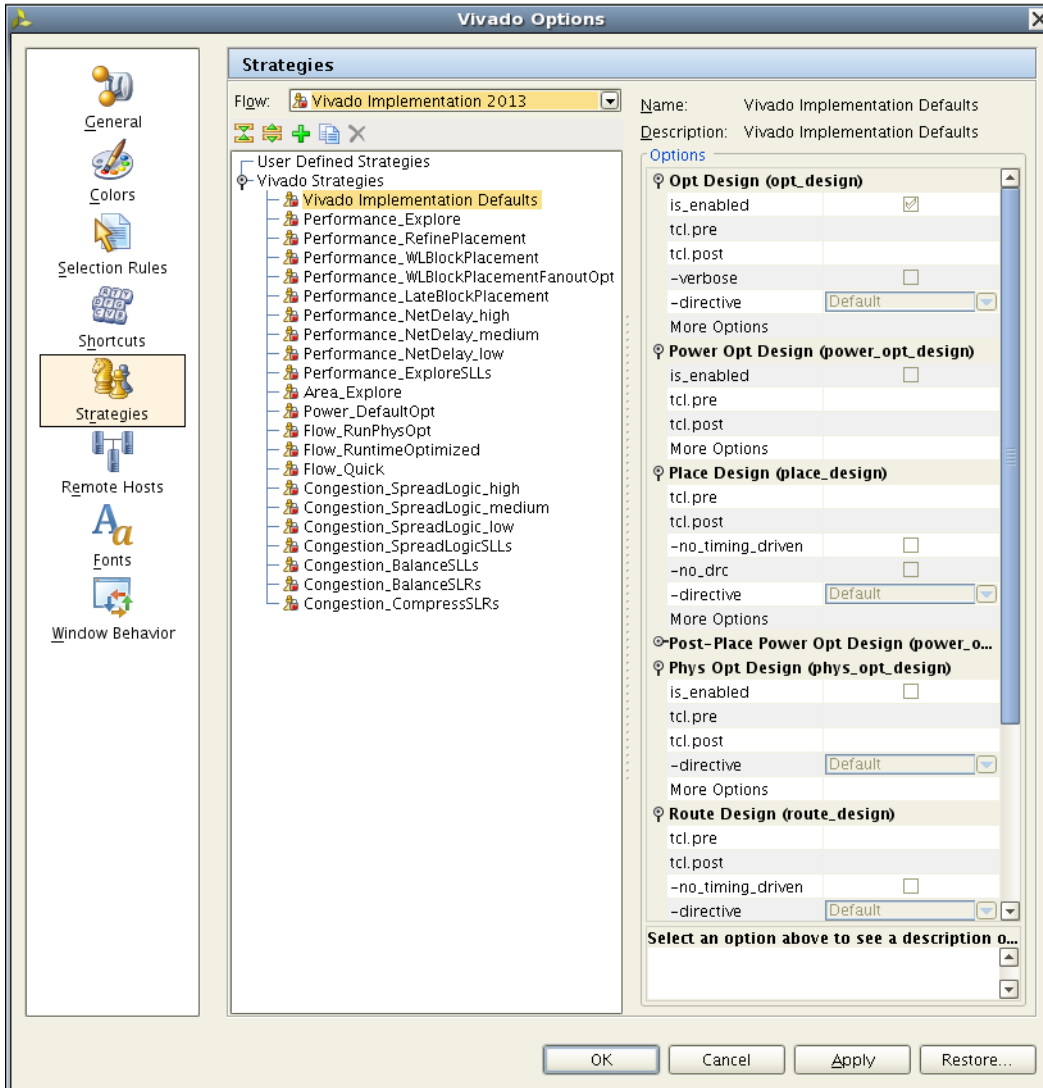


Figure 1-11: Default Implementation Strategies

Vivado implementation includes several commonly used strategies that are tested against internal benchmarks.



TIP: You cannot save changes to the predefined implementation strategies. However, you can copy, modify, and save the predefined strategies to create your own.

Accessing Currently Defined Strategies

To access the currently defined strategies, select **Tools > Options** in the Vivado IDE main menu.



Figure 1-11, [Default Implementation Strategies](#), shows the default strategies included with the Vivado tools.

Reviewing, Copying, and Modifying Strategies

To review, copy, and modify strategies:

1. Select **Tools > Options** from the main menu.
2. Select **Strategies** in the left-side panel.

The Strategies dialog box contains a list of pre-defined strategies for various tools and release versions. See [Figure 1-11, Default Implementation Strategies](#).

3. In the **Flow** pulldown, select the appropriate **Vivado Implementation** version for the available strategies. A list of included strategies is displayed.
4. To create a new strategy, select **Create New Strategy** on the toolbar or from the  popup menu.
5. To copy an existing strategy, select **Create a copy of this strategy** from the toolbar or from the popup menu. The Vivado design tool:
 - a. Creates a copy of the currently selected strategy.
 - b. Adds it to the User Defined Strategies list.
 - c. Displays the strategy options on the right side of the dialog box for you to modify. 
6. Provide a name and description for the new strategy as follows:
 - **Name**
Enter a strategy name to assign to a run.
 - **Type**
Specify **Synthesis** or **Implement**.
 - **Tool Version**
Specify the tool version.
 - **Description**
Enter the strategy description displayed in the Design Run results table.

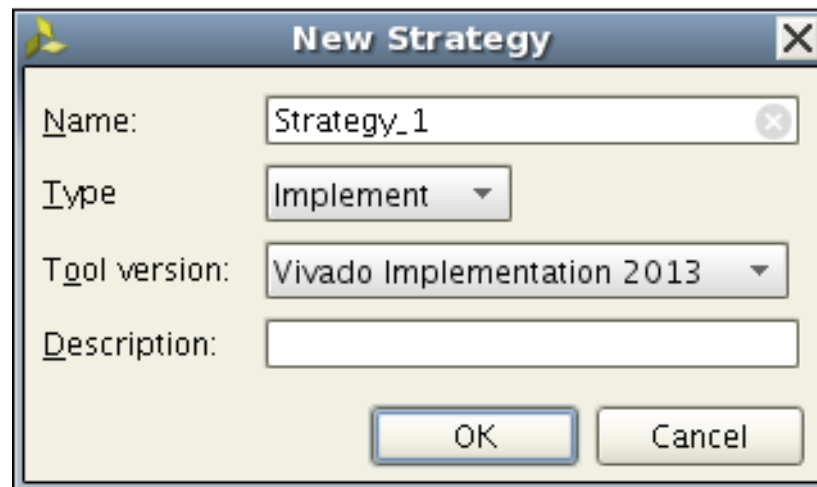


Figure 1-12: New Strategy Dialog Box

7. Edit the **Options** for the various implementation steps:
 - **opt_design**
 - **power_opt_design**
 - **place_design**
 - **phys_opt_design**
 - **route_design**
 - **write_bitstream**



TIP: Select an option to view a brief description of the option at the bottom of the Design Run Settings dialog box.

For more information about the implementation steps and their options, see [Chapter 2, Implementation Commands](#).

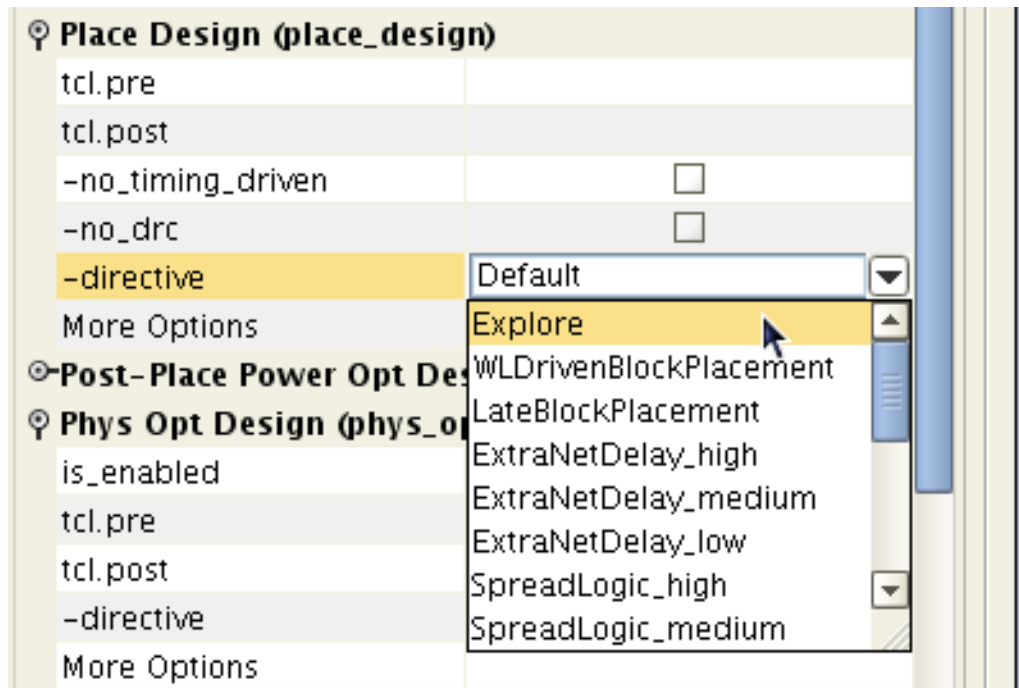


Figure 1-13: Edit Implementation Steps

8. Click the right-side column of a specific option to modify command options. See [Figure 1-13, Edit Implementation Steps](#), immediately above for an example.

You can then:

- Select predefined options from the pull down menu.
- Enable or disable some options with a check box.
- Type a user-defined value for options with a text entry field.
- Use the file browser to specify a file for options accepting a file name and path.
- Insert a custom Tcl script (called a hook script) before and after each step in the implementation process (`tcl.pre` and `tcl.post`). This lets you perform specific tasks either before or after each implementation step (for example, generating a timing report before and after Place Design to compare timing results).

For more information on defining Tcl hook scripts, see the *Vivado Design Suite User Guide: Using Tcl Scripting (UG894)* [Ref 3].

Note: Relative paths in the `tcl.pre` and `tcl.post` scripts are relative to the appropriate run directory of the project they are applied to: `<project>/<project.runs>/<run_name>`

You can use the `DIRECTORY` property of the current project or current run to define the relative paths in your scripts:

```
get_property DIRECTORY [current_project]
get_property DIRECTORY [current_run]
```

- Click **OK** to save the new strategy.

The new strategy is listed under User Defined Strategy. The Vivado tool saves user-defined strategies to the following locations:

- Linux OS

```
$HOME/.Xilinx/Vivado/strategies
```

- Windows 7

```
C:\Users\\AppData\Roaming\Xilinx\Vivado\strategies
```

- Windows XP

```
C:\Documents and Settings\username\Application Data\Xilinx\Vivado\strategies
```

Sharing Strategies

Design teams that want to create and share strategies can copy any user-defined strategy from the user directory to the `<InstallDir>/Vivado/<version>/strategies` directory

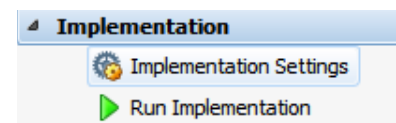
where

- `<InstallDir>` is the installation directory of the Xilinx software.
- `<version>` is the release version.

Launching Implementation Runs

Do any of the following to launch the active implementation run in the Design Runs window:

- Select **Run Implementation** in the Flow Navigator.
- Select **Flow > Run Implementation** from the main menu.
- Select **Run Implementation** from the toolbar menu.
- Select a run in the Design Runs window and select **Launch Runs** from the popup menu.



Launching a single implementation run initiates a separate process for the implementation.



TIP: Select a run in the Design Runs window to launch a run other than the active run. Select two or more runs in the Design Runs window to launch multiple runs at the same time.

1. Use **Shift+click** or **Ctrl+click** to select multiple runs.

Note: You can choose both synthesis and implementation runs when selecting multiple runs in the Design Runs window. The Vivado IDE manages run dependencies and launches runs in the correct order.

2. Select **Launch Runs** to open the Launch Selected Runs dialog box. See [Figure 1-14, Launch Selected Implementation Runs](#).

Note: You can select **Launch Runs** from the popup menu, or from the Design Runs window toolbar menu

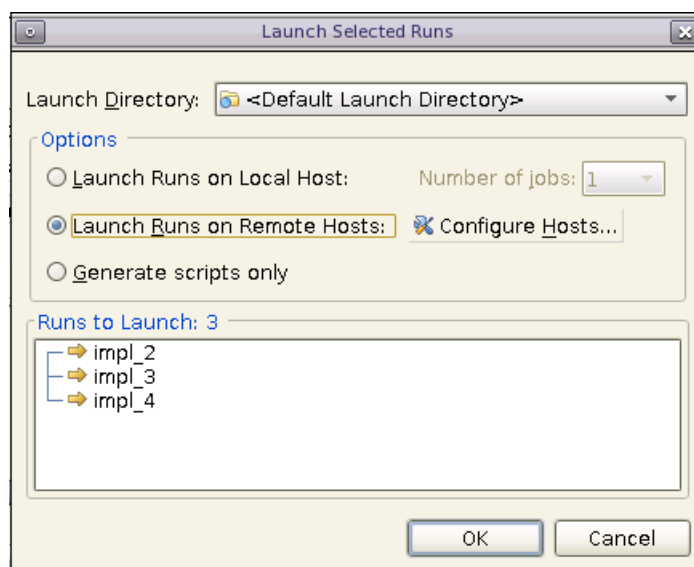


Figure 1-14: Launch Selected Implementation Runs

3. Select **Launch Directory**.

The default launch directory is in the local project directory structure. Files for implementation runs are stored at:

```
<project_name>/<project_name>.runs/<run_name>
```



TIP: Defining any non-default location outside the project directory structure makes the project non-portable because absolute paths are written into the project files.

4. Specify **Options**.

a. **Launch Runs on Local Host**

Launch the run on the local machine.

b. **Number of Jobs**

Define the number of local processors to use when launching multiple runs simultaneously. Individual runs are launched on each processor.

c. **Launch Runs on Remote Hosts** (Linux only)

Use remote hosts to launch one or more jobs.

For more information, see [Appendix A, Using Remote Hosts](#).

d. **Configure Hosts**

Select this option to configure remote hosts.

e. **Generate scripts only**

Export and create the run directory and run script, but do not launch the run at this time. You can run the script later outside the Vivado tool.

Moving Processes into the Background

As the Vivado IDE initiates the process to run synthesis or implementation, it reads design files and constraint files in preparation for the run. The Starting Run dialog box lets you move this preparation into the background. See [Figure 1-15, Start Run - Background Process](#).

Putting this process into the background releases the Vivado IDE to perform other functions while it completes the background task. The other functions can include functions such as viewing reports and opening design files. You can use this time, for example, to review previous runs, or to examine reports.



CAUTION! *When you put this process into the background, the Tcl Console is blocked. You cannot execute Tcl commands, or perform tasks that require Tcl commands, such as switching to another open design.*

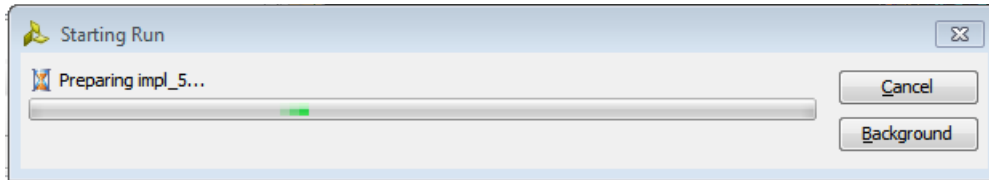


Figure 1-15: Start Run - Background Process

Running Implementation in Steps

Vivado implementation consists of a number of smaller processes such as:

- Logic optimizer
- Placer
- Router

The Vivado tools let you run implementation as a series of steps, rather than as a single process.

How to Run Implementation in Steps

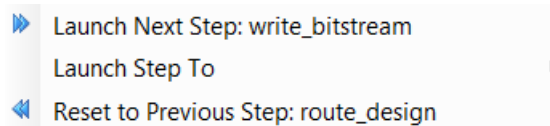
To run implementation in steps:

1. Select a run in the Design Runs window.
2. Select **Launch Next Step: <Step>** or **Launch Step To** from the popup menu.

Valid **<Step>** values depend on which run steps have been enabled in the Run Settings. The steps that are available in an implementation run are:

- **Opt Design**
Optimizes the logical design and fit sit onto the target Xilinx FPGA device.
- **Power Opt Design**
Optimizes elements of the design to reduce power demands of the implemented FPGA device.
- **Place Design**
Places the design onto the target Xilinx device.
- **Phys Opt Design**
Performs timing-driven optimization on the negative-slack paths of a design.

- **Route Design**
Routes the design onto the target Xilinx device.
 - **Write Bitstream**
Generates a bitstream for Xilinx device configuration. Although not technically part of an implementation run, bitstream generation is available as an incremental step.
3. Repeat **Launch Next Step: <Step>** or **Launch Step To** as needed to move the design through implementation.



You can run any reports or analysis needed between implementation steps to explore design options.

4. To back up from a completed step, select **Reset to Previous Step: <Step>** from the Design Runs window popup menu.

Select **Reset to Previous Step** to reset the selected run from its current state to the prior incremental step. This allows you to:

- Step backward through a run.
- Make any needed changes.
- Step forward again to incrementally complete the run.

Monitoring the Implementation Run

Monitoring the implementation run allows you to:

- Read the compilation information.
- Review warnings and errors in the Messages window.
- View the Project Summary.
- Open the Design Runs window.

Monitor the status of a Synthesis or Implementation run in the Log window.

Viewing the Run Status Display

The status of a run that is in progress can be displayed in two ways for synthesis and implementation runs. These status displays show that a run is in progress. They allow you to cancel the run if desired. See [Figure 1-16, Implementation Run Status](#).

Design Runs			
Name	Part	Constraints	Strategy
synth_1	xc7k70tfbg484-2	constrs_2	Vivado Synthesis Defaults (Vivado Synthesis 2013)
impl_1	xc7k70tfbg484-2	constrs_2	Vivado Implementation Defaults (Vivado Implementation 2013)

Figure 1-16: Implementation Run Status

The Run Status indicator in the project status bar at the upper right corner of the Vivado IDE displays a scrolling bar to indicate the run is in process. You can use the **Cancel** button to end the run.

The Run Status indicator in the Design Runs window, as shown at the bottom of [Figure 1-16, Implementation Run Status](#), displays a circular arrow to indicate the run is in process. You can select the run and use the Reset Run command from the popup menu to cancel the run.

Cancelling/Resetting the Run

If you cancel a run that is in-progress, either through the **Cancel** button, or through the **Reset Run** command, the Vivado IDE prompts you to delete any run files created during the cancelled run. See [Figure 1-17, Cancel Implementation](#).

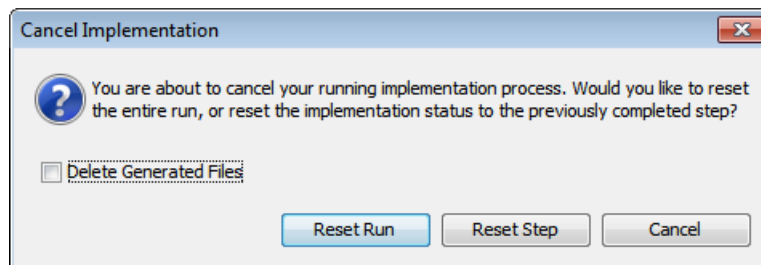


Figure 1-17: Cancel Implementation

Select **Delete Generated Files** to clear the run data from the local project directories.



RECOMMENDED: Delete any data created as a result of a cancelled run to avoid conflicts with future runs.

Viewing the Log in the Log Window

The Log window opens in the Vivado IDE after you launch a run. It shows the standard output messages. It also displays details about the progress of each individual implementation process, such as **place_design** and **route_design**.

The Log window can help you understand where different messages originate to aid in debugging the implementation run. See [Figure 1-18, Compilation Log](#), for an example of the Log window.



TIP: View the Log window to see where messages originated as an aid in debugging the implementation run.

```
Log
Starting Placer Task
INFO: [Place 30-611] Multithreading enabled for place_design using a maximum of 4 CPUs
Phase 1 Mandatory Logic Optimization
Netlist sorting complete. Time (s): cpu = 00:00:00.06 ; elapsed = 00:00:00.06 . Memory (MB): peak = 923.789 ; gain = 0.000
Phase 1 Mandatory Logic Optimization | Checksum: 941e5879
Time (s): cpu = 00:00:00.43 ; elapsed = 00:00:00.44 . Memory (MB): peak = 923.789 ; gain = 0.000
Phase 2 Build SLR Info
Phase 2 Build SLR Info | Checksum: 941e5879
```

Synthesis | **Implementation** | Simulation

Figure 1-18: Compilation Log

Pausing Output

Click **Pause** to pause the output to the Log window. Pausing allows you to read the log while implementation continues running.



Displaying the Project Status

The Vivado IDE uses several methods to display the project status and which step to take next. The project status reports only the results of the major design tasks.

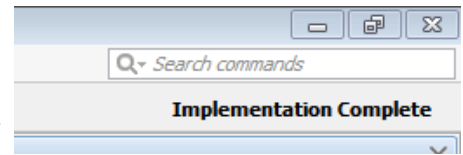
The project status is displayed in the Project summary and the Status bar. It allows you to immediately see the status of a project when you open the project, or while you are running the design flow commands, including:

- RTL elaboration
- Synthesis
- Implementation
- Bitstream generation

Viewing Project Status in the Project Status Bar

The project status is displayed in the project status bar in the upper-right corner of the Vivado IDE.

As the run progresses through the Synthesize, Implement, and Write Bitstream commands, the Project Status Bar changes to show either a successful or failed attempt. Failures are displayed in red text.



Viewing Out-of-Date Status

If source files or design constraints change, and either synthesis or implementation was previously completed, the project might be marked as **Out-of-Date**. See [Figure 1-19, Implementation Out-of-Date](#).

The project status bar shows an Out-of-Date status. Click **more info** to display which aspects of the design are out of date. It may be necessary to rerun implementation, or both synthesis and implementation.

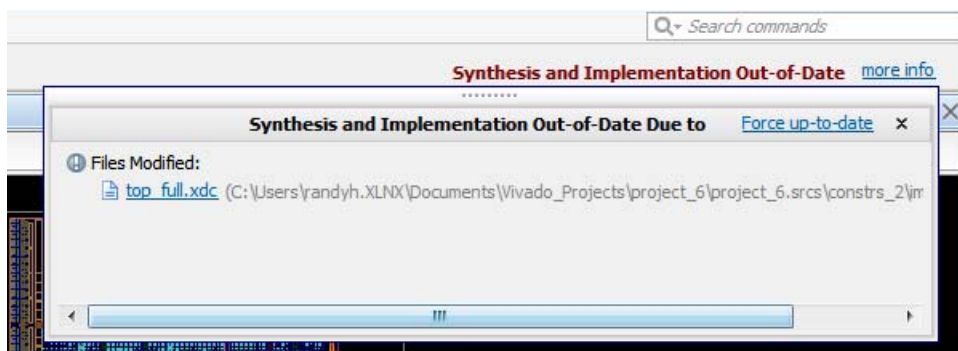


Figure 1-19: Implementation Out-of-Date

Forcing Runs Up-to-Date

Click **Force-up-to-date** to force the implementation or synthesis runs up to date. Use **Force-up-to-date** if you changed the design or constraints, but still want to analyze the results of the current run.



TIP: The **Force-up-to-date** command is also available from the popup menu of the Design Runs window when an out-of-date run is selected.

For more information, see the *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)* [Ref 2].

Saving Placer and Router Runtime with Incremental Compile

Incremental Compile is an advanced design flow for designs that are nearing completion, and that require significant investment in timing closure. After resynthesizing small changes, the flow: (1) speeds up place and route runtime; and (2) preserves timing closure by reusing prior placement and routing from a reference design. The flow is most effective when synthesis changes result in at least 95 percent similarity to the reference design.

Note: Only the changed portions of the current design are placed and routed from scratch.

Incremental Compile Flow Diagram

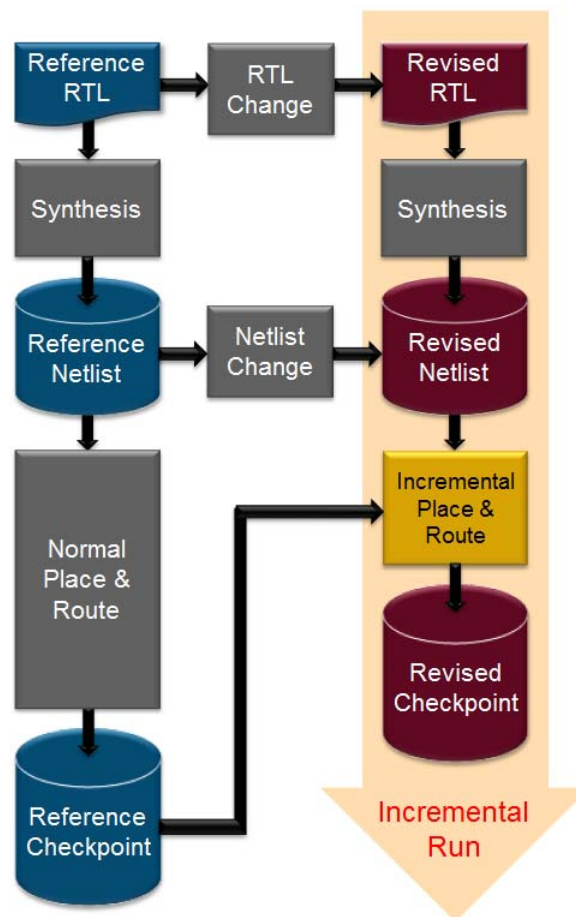


Figure 1-20: Incremental Compile Flow

Incremental Compile Flow Designs

The Incremental Compile flow involves two different designs:

- [Reference Design](#)
- [Current Design](#)

See [Figure 1-20, Incremental Compile Flow](#).

Reference Design

The first Incremental Compile flow design is the *reference* design.

The reference design is usually an earlier iteration or variation of the current design that has been synthesized, placed, and routed.

The reference design checkpoint (DCP) may be the product of many design iterations involving code changes, floorplanning, and revised constraints necessary to close timing.

After the current design is loaded, load the reference design checkpoint using the **read_checkpoint -incremental <dcg>** command. Loading the reference design checkpoint with the **-incremental** option enables the Incremental Compile design flow for subsequent place and route operations. For more information, see [Examining the Similarity Between the Reference Design and the Current Design, page 50](#).

Current Design

The second Incremental Compile flow design is the *current* design.

The current design incorporates small design changes or variations from the reference design. These changes or variations can include:

- RTL changes
- Netlist changes
- Both RTL changes and netlist changes

Running Incremental Place and Route

The updated, current design is first loaded into memory. The reference design checkpoint is then loaded incrementally.

The key component of the Incremental Compile process is incremental place and route. The reference design checkpoint contains a netlist, constraints, and physical data including the placement and routing.

- The netlist in the current design is compared to the reference design to identify matching cells and nets.
- Placement from the reference design checkpoint is reused to place matching cells in the current design.
- Routing is reused to route matching nets.

Nets with Multiple Fanouts

The Vivado router performs fine-grained matching for nets with multiple fanouts, in which each routing segment can be reused or discarded as appropriate.

Design objects that do not match between the reference design and the current design are placed after incremental placement is complete, and routed after existing routing is reused.

Effectively Reusing Placement and Routing

Effective reuse of the placement and routing from the reference design depends on the design differences between the two designs. Even small differences can have a large impact.

Impact of Small RTL Changes

Although synthesis tries to minimize netlist name changes, sometimes small RTL changes such as the following can lead to very large design changes:

- Increasing the size of an inferred memory
- Widening an internal bus
- Changing data types from unsigned to signed

Impact of Changing Constraints and Synthesis Options

Similarly, changing constraints and synthesis options such as the following can also have a large impact on incremental placement and routing:

- Changing timing constraints and resynthesizing
- Preserving or dissolving logical hierarchy
- Enabling register re-timing

Examining the Similarity Between the Reference Design and the Current Design

Run **report_incremental_reuse** to examine and report the similarity between a reference design checkpoint file and the current design. The **report_incremental_reuse** command compares the netlist from the reference design checkpoint with the current in-memory design, and reports the percentage of matching of cells, nets, and ports.

Note: The **report_design_similarity** command in previous versions of the Vivado tool has been replaced with the **report_incremental_reuse** command in the current version. The new **report_incremental_reuse** command (unlike the old **report_design_similarity** command) does not take an argument. Instead you must run **read_checkpoint -incremental <dcg>** first. You can then run **report_incremental_reuse** at any time thereafter. It automatically compares the design in memory against the **<dcg>** that was read in by **read_checkpoint -incremental**.

A higher degree of design similarity results in more effective reuse of placement and routing from the reference design. Below is an example of a **report_incremental_reuse** report comparing an in-memory synthesized design to a **route_design** checkpoint

Type	Count	Total	Percentage
Reused Cells	3591	3821	93.98
Reused Ports	71	71	100.00
Reused Nets	6142	6564	93.57
Reused Cells	3591		
Non-Reused Cells	230		
New	230		
Fully reused nets	5383		
Partially reused nets	759		
New nets	422		

In general, the higher the match percentage, the more placement and routing can be reused, and the faster place and route will run.



RECOMMENDED: Select **report_incremental_reuse** to assess the feasibility of using a checkpoint as a reference design for running incremental place and route on the current design.

Incremental Place and Route Metrics

Incremental place and route in the Vivado Design Suite can achieve an average of a threefold improvement over normal place and route runtimes when designs have at least 95 percent similar cells, nets, and ports.

The average runtime improvement decreases as similarity between the reference design and the current design decreases.

Below 80 percent, there may be little or no benefit to using the incremental place and route feature.

Factors Affecting Runtime Improvement

Other factors that can affect runtime improvement include:

- The amount of change in timing-critical areas. If critical path placement and routing cannot be reused, more effort is required to meet timing. In other words, the performance benefits of incremental place and route is wasted on the simpler portions of the design.
- The **phys_opt_design** command performs timing-driven logic transformations that may produce different results when run in the Incremental flow or standard flow. If design changes are concentrated in timing-critical areas that are affected by **phys_opt_design**, the reuse of routes may not provide the expected benefit.
- The initialization portion of the place and route runtime. In short place and route runs, the initialization overhead of the Vivado placer and router may eliminate any gain from the incremental place and route process. For designs with longer runtimes, initialization becomes a small percentage of the runtime.

Using Incremental Compile

In both Project Mode and Non-Project Mode, incremental place and route mode is entered when you load the reference design checkpoint using the **read_checkpoint -incremental <dcp_file>** command where *<dcp_file>* specifies the path and filename of the reference design checkpoint. Loading the reference design checkpoint with the **-incremental** option enables the Incremental Compile design flow for subsequent place and route operations. In Non-Project Mode, **read_checkpoint -incremental** should: (1) *follow* **opt_design** and; (2) *precede* **place_design**.

To exit the incremental compile mode:

1. Read a checkpoint without the **-incremental** option.

OR
2. If using projects, remove the setting of the Incremental Compile checkpoint.

To specify a design checkpoint file (DCP) to use as the reference design, and to run incremental place:

1. Load the current design.
2. Run **opt_design**.
3. Run **read_checkpoint -incremental <dcp_file>**.
4. Run **place_design**.

```
link_design; # to load the current design
opt_design
read_checkpoint -incremental <dcp_file>
place_design
```

Incremental placement relies on the ability of the Vivado tool to match design objects in the current design with design objects in the reference design. However, the **opt_design -resynth_area** command option (or **-directive ExploreArea**) usually results in poor cell-name matching between the reference and current designs. This in turn results in poor matching of placement and routing data.



RECOMMENDED: *When running the Incremental Compile flow, limit the use of **opt_design -resynth_area**.*

Run the **read_checkpoint -incremental <dcp_file>** command followed by **route_design** to specify a design checkpoint file (DCP) to use as the reference design and run incremental route. If an incremental checkpoint has already been read in the previous **place_design** step: (1) Vivado is still in incremental mode; and (2) reading the checkpoint with the **-incremental** option is not needed before **route_design**.

During routing, the route connections from the reference design are applied as the checkpoint is read.

Orphaned Route Segments

Some cells may have been eliminated from the current design, or moved during placement, leaving orphaned route segments from the reference design. If you are running in the Vivado IDE, you may see potentially problematic nets. These orphaned or improperly connected route segments are cleaned up during incremental routing by the Vivado router.

The following INFO message appears during placement.

```
INFO: [Place 46-2] During incremental compilation, routing data from the original
checkpoint is applied during place_design. As a result, dangling route segments and
route conflicts may appear in the post place_design implementation due to changes
between the original and incremental netlists. These routes can be ignored as they
will be subsequently resolved by route_design. This issue will be cleaned up
automatically in place_design in a future software release.
```

Using Incremental Compile in Project Mode

In Project Mode, you can set the incremental compile option in the Design Runs window.

To set the incremental compile option:

1. Select a run in the Design Runs window.
2. Click **Set Incremental Compile** from the context menu.
3. In the Set Incremental Compile window, select a reference design checkpoint.

This enables incremental compile mode for the run.



IMPORTANT: When you choose a checkpoint from a design run, it is deleted if the design run is reset. If you want to choose a checkpoint from a design run, copy it into a separate directory before selecting it as your reference checkpoint.

See [Figure 1-21, Set Incremental Compile on Design Run](#).

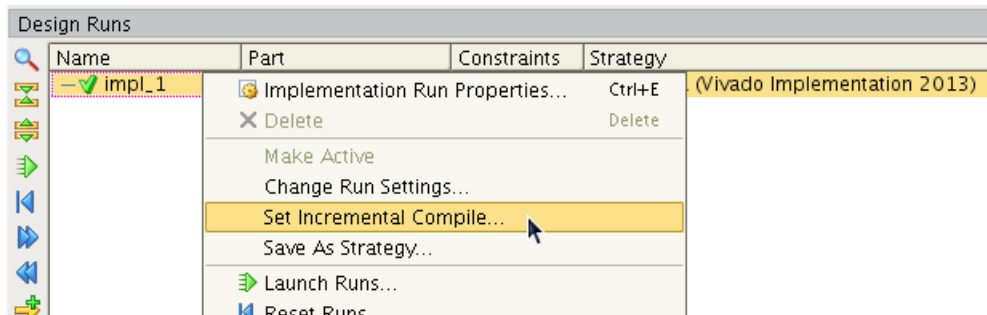


Figure 1-21: Set Incremental Compile on Design Run

To disable incremental compile mode for the run, leave the Use Checkpoint field blank. See [Figure 1-22, Setting Incremental Compile](#).

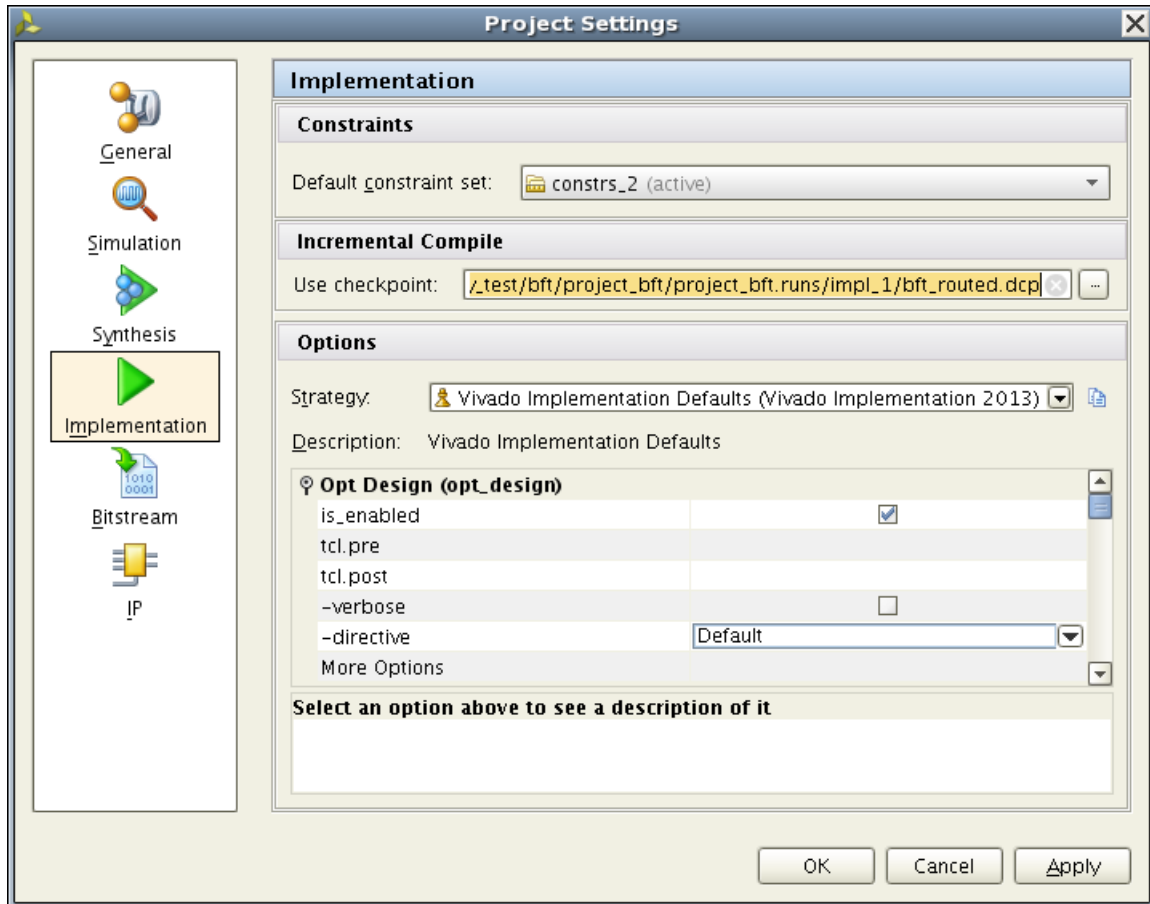


Figure 1-22: Setting Incremental Compile

After incremental placement and routing are complete, you can review the messages generated by the Vivado tool.

Incremental Compile Advanced Analysis

After completing an incremental place and route, you can analyze timing with details of cell and net reuse. Objects are tagged in timing reports to show the level of physical data reuse. This identifies whether or not your design updates are affecting critical paths.

To see incremental flow details in timing reports, use the **report_timing -label_reused** option. This generates a report showing reuse labels on input and output pins, indicating the amount of physical data reused for the pin's cell and net.

- **(R)**: Both the cell placement and net routing are reused.
- **(NR)**: Neither the cell placement nor the net routing are reused.
- **(PNR)**: The cell placement is reused but the net routing is not reused.
- **(N)**: The pin, cell, or net is a new design object, not present in the reference design.

See the following example.

```

-----
(NR)SLICE_X46Y42  FDRE (Prop_fdre_C_Q)  0.259  -1.862  r  fftI/fifoSel_reg[5]/Q
                    net (fo=8, estimated)  0.479  -1.383  r  fftI/n_fifoSel_reg[5]
(R)SLICE_X46Y43
(R)SLICE_X46Y43  LUT4 (Prop_lut4_I1_O)  0.043  -1.340  r  fftI/wbDOut_reg[31]i5/I1
                    net (fo=32, routed)  1.325  -0.014  r  fftI/wbDOut_reg[31]i5/O
                    r  fftI/wbDOut_reg[31]i5
(R)SLICE_X44Y39
(PNR)SLICE_X44Y39  MUXF7 (Prop_muxf7_S_O)  0.154  0.140  r  fftI/wbcI/wbDOut_reg[0]i1/S
fftI/wbcI/wbDOut_reg[0]i1/O
                    net (fo=1, routed)  0.000  0.140  r  fftI/wbcI/wbDOut_reg[0]i1
(PNR)SLICE_X44Y39  r  fftI/wbDout_reg[0]/D
-----

```

The **read_checkpoint -incremental** command assigns two cell properties which are useful for analyzing incremental flow results using scripts or interactive Tcl commands.

- **IS_REUSED:** A boolean property which is TRUE if the cell's placement is reused from the reference design
- **REUSE_STATUS:** A string property denoting the cell's reuse status after incremental placement and routing. Possible values are:
 - New
 - Reused
 - Discarded to improve timing

Moving Forward After Implementation

After implementation has completed, for both Project Mode and Non-Project Mode, the direction you take the design next depends on the results of the implementation.

- Is the design fully placed and routed, or are there issues that need to be resolved?
- Have the timing constraints and design requirements been met, or are their additional changes required to complete the design?
- Are you ready to generate the bitstream for the Xilinx part?

Recommended Steps After Implementation

The recommended steps after implementation are:

1. Review the implementation messages.
2. Review the implementation reports to validate key aspects of the design:
 - Timing constraints are met (**report_timing_summary**).
 - Utilization is as expected (**report_utilization**).
 - Power is as expected (**report_power**).
3. Write the bitstream file.

Writing the bitstream file includes a final DRC to ensure that the design does not violate any hardware rules.

4. If any design requirements have not been met:
 - a. In Project Mode, open the implemented design for further analysis.
 - b. In Non-Project Mode, open a post-implementation design checkpoint.

For more information on analysis of the implemented design, see the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques (UG906)* [Ref 8].

Moving Forward in Non-Project Mode

In Non-Project Mode, the Vivado Design Suite generated messages for the design session, and wrote the messages to the Vivado log file (`vivado.log`). Examine this log file and the reports generated from the design data to view an accurate assessment of the current project state.

Moving Forward in Project Mode

In Project Mode, the Vivado Design Suite:

- Displays the messages from the log file in the Messages window.
- Automates the creation and delivery of numerous reports for you to review.

In Project Mode, after an implementation run is complete in the Vivado IDE, you are prompted for the next step. See [Figure 1-23, Project Mode - Implementation Completed](#).

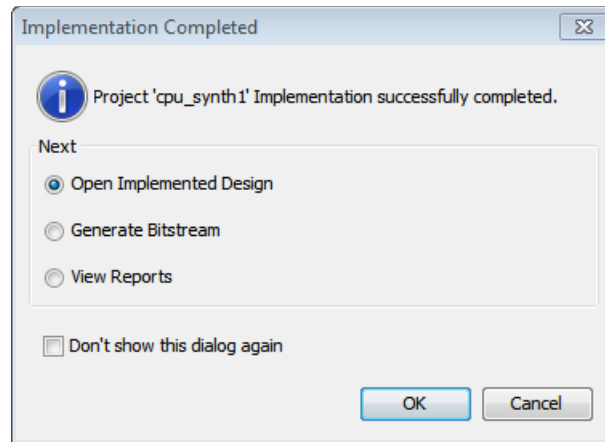


Figure 1-23: Project Mode - Implementation Completed

In the Implementation Completed dialog box:

1. Select the appropriate option:

- **Open Implemented Design**

Imports the netlist, design constraints, the target part, and the results from place and route into the Vivado IDE for design analysis and further work as needed.

- **Generate Bitstream**

- Launches the Generate Bitstream dialog box.
- For more information, see the *Vivado Design Suite User Guide: Programming and Debugging (UG908)* [Ref 10].

- **View Reports**

- Opens the Reports window for you to select and view the various reports produced by the Vivado tools during implementation.
- For more information, see [Viewing Implementation Reports, page 60](#).

2. Click **OK**.

Viewing Messages



IMPORTANT: Review all messages. The messages may suggest ways to improve your design for performance, power, area, and routing. Critical warnings may also expose timing constraint problems that must be resolved.

Viewing Messages in Non-Project Mode

In Non-Project Mode, review the Vivado log file (`vivado.log`) for:

- The commands that you used during a single design session
- The results and messages from those commands



RECOMMENDED: Open the log file in the Vivado text editor and review the results of all commands for valuable insights.

Viewing Messages in Project Mode

In Project Mode, the Messages window displays a filtered list of the Vivado log. This list includes only the main Messages, Warnings, and Errors. The Messages window sorts by feature, and includes toolbar options to filter and display only specific types of messages.

See [Figure 1-24, Messages Window](#), for an example.

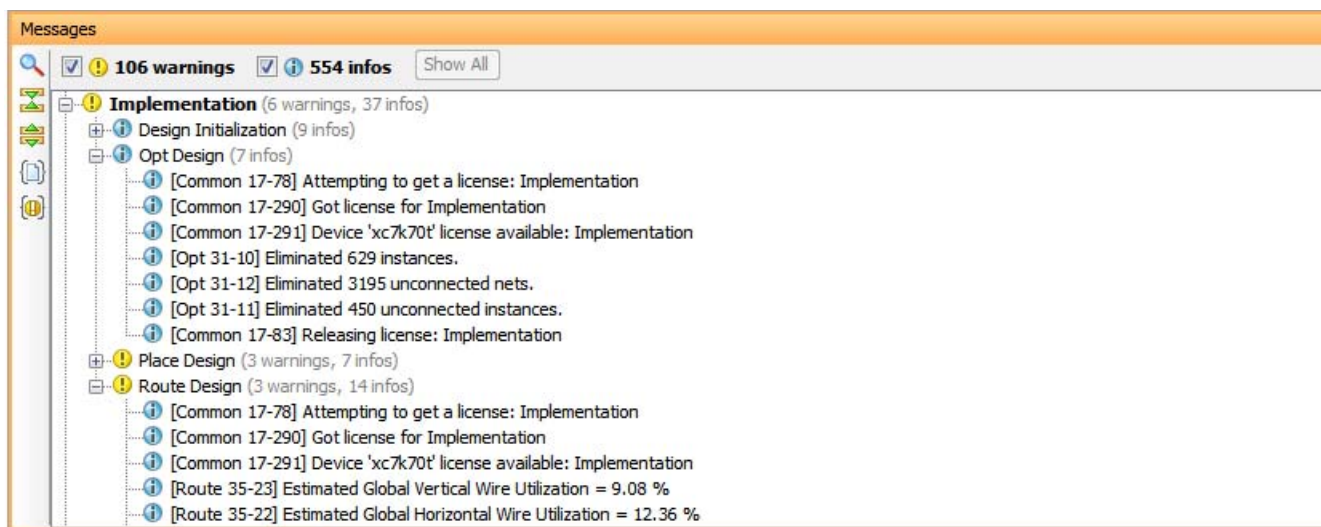



Figure 1-24: Messages Window

Use the following features when viewing messages in Project Mode:

- Click the expand and collapse tree widgets to view the individual messages. 
- Check the appropriate check box in the banner to display Errors, Critical Warnings, Warnings, and Informational Messages in the Messages window.
- Select a linked message in the Messages window to open the source file and highlight the appropriate line in the file.
- Run **Search for Answer Record** from the Messages window popup menu to search the Xilinx Customer Support database for answer records related to a specific message.

Incremental Compile Messages

The Vivado tool reports summary results from Incremental Compile in the log file, including:

- [Incremental Placement Summary](#)
- [Incremental Routing Summary](#)

Incremental Placement Summary

The following example of the Incremental Placement Summary includes a final assessment of:

- Cell placement reuse
- Runtime statistics

```

+-----+
| Incremental Placement Summary |
+-----+
| Reused instances | 40336 |
| Non-reused instances | 1158 |
| %similarity | 97.21 |
+-----+
| Incremental Placement Runtime Summary |
+-----+
| Initialization time(elapsed secs) | 87.54 |
| Incremental Placer time(elapsed secs) | 50.42 |
+-----+

```

Incremental Routing Summary

The Incremental Routing Summary displays reuse statistics according to net fanout distribution. The categories reported include:

- **Fully Reused**

The entire net's routing is reused from the reference design.

- **Partially Reused**

Some of the net's routing from the reference design is reused. Some segments are re-routed due to changed cells, changed cell placements, or both.

- **New/Unmatched**

The net in the current design was not matched in the reference design.

```

+-----+
| Incremental Routing Reuse Summary:
+-----+-----+-----+-----+-----+
| #      | Fully | Partially | New / |      |
| # Fanout | Reused | Reused   | Unmatched | Total |
+-----+-----+-----+-----+-----+
| == 2    | 15749 | 12       | 387    | 16136 |
+-----+-----+-----+-----+-----+
| < 50    | 16061 | 31       | 375    | 16467 |
+-----+-----+-----+-----+-----+
| [50,100] | 249   | 3        | 20     | 272   |
+-----+-----+-----+-----+-----+
| [100,500] | 32   | 0        | 7      | 39    |
+-----+-----+-----+-----+-----+
| >= 500  | 8     | 1        | 0      | 9     |
+-----+-----+-----+-----+-----+
| Total   | 32099 | 47       | 789    | 32923 |
+-----+-----+-----+-----+-----+

```

```

+-----+-----+-----+
| %      | Fully | Partially |
| Reuse  | Reused | Reused   |
+-----+-----+-----+
| Net    | 97.49 | 0.14     |
| Pin    | 100.00 | 0.00    |
+-----+-----+-----+

```

Viewing Implementation Reports

The Vivado Design Suite generates many types of reports, including reports on:

- Timing, timing configuration, and timing summary
- Clocks, clock networks, and clock utilization
- Power, switching activity, and noise analysis

When viewing reports, you can:

- Browse the report file using the scroll bar.
- Click **Find** or **Find in Files** to search for specific text.



- Click **Go to the Beginning** to scroll to the beginning file.



- Click **Go to the End** to scroll to end of the file.



Reporting in Non-Project Mode

In Non-Project Mode, you must run these reports manually.

- Use Tcl commands to create an individual report.
- Use a Tcl script to create a series of reports.

Example Tcl Script

The following Tcl script runs a series of reports and saves them to a Reports folder:

```
# Report the control sets sorted by clk, clkEn
report_control_sets -verbose -sort_by {clk clkEn} -file C:/Report/cntrl_sets.rpt
# Run Timing Summary Report for post implementation timing report_timing_summary
-file C:/Reports/post_route_timing.rpt -name time1
# Run Utilization Report for device resource utilization report_utilization -file
C:/Reports/post_route_utilization.rpt
```

Opening Reports in a Vivado IDE Window

You can open these reports in a Vivado IDE window. In the [Example Tcl Script](#) immediately above, the `report_timing_summary` command:

- Uses the `-file` option to direct the output of the report to a file.
- Uses the `-name` option to direct the output of the report to a Vivado IDE window.

[Figure 1-26, Control Sets Report](#), shows an example of a report opened in a Vivado IDE window.



TIP: The directory to which the reports are to be written to must exist before running the report, or the file cannot be saved, and an error message will be generated.

Getting Help With Implementation Reports

Use the Tcl **help** command in the Vivado IDE or at the Tcl command prompt.

For a complete description of the Tcl reporting commands and their options, see the *Vivado Design Suite Tcl Command Reference Guide (UG835)* [Ref 12].

Reporting in Project Mode

In Project Mode, many reports are generated automatically. View report files in the Reports window. See [Figure 1-25, Reports Window](#).

The Reports window usually opens automatically after synthesis or implementation commands are run. If the window does not open:

- Select the **Reports** link in the Project Summary, or
- Select **Windows > Reports**.

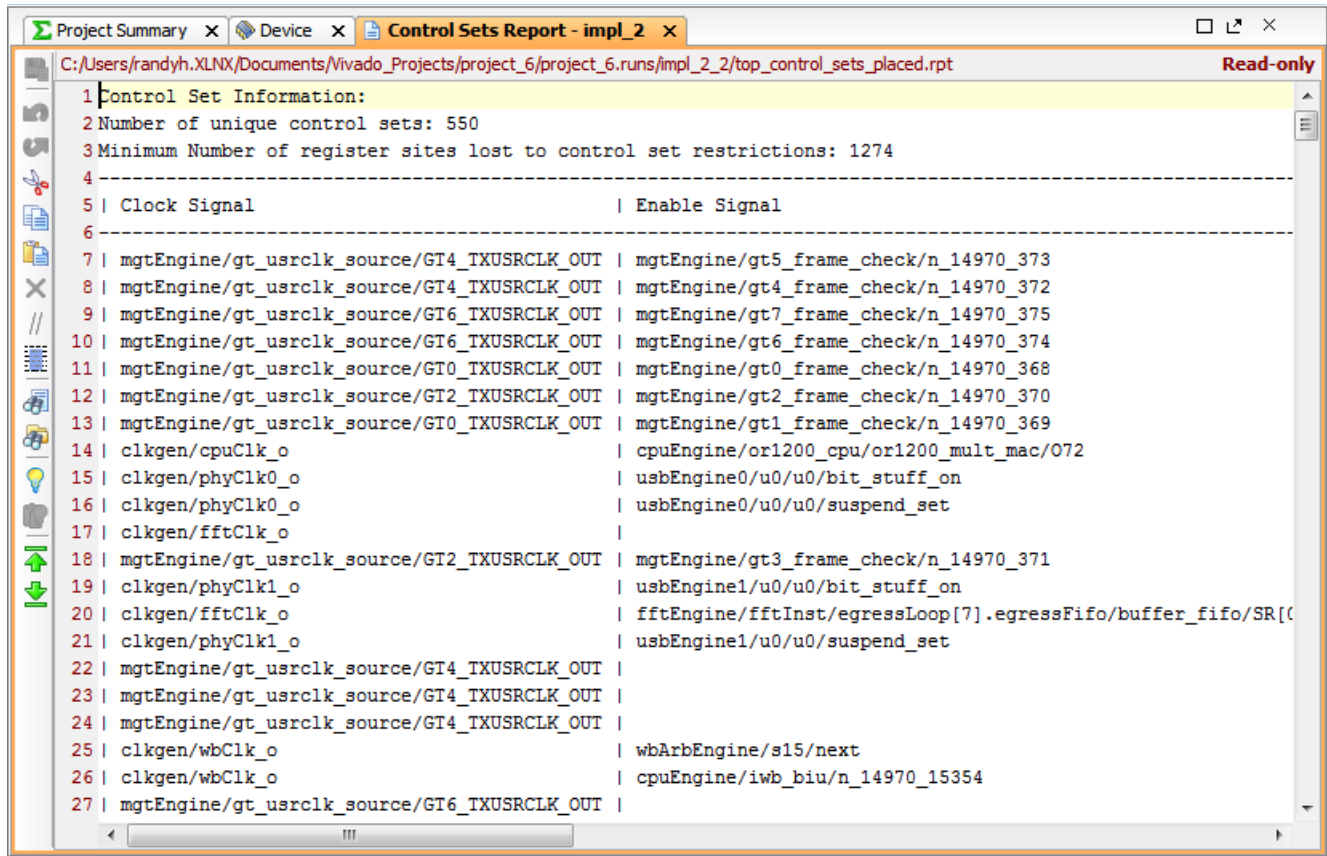


TIP: The **tcl.pre** and **tcl.post** options of an implementation run let you output custom reports at each step in the process. These reports are not listed in the Reports window, but can be customized to meet your specific needs. For more information, see [Changing Implementation Run Settings](#).

Name	Modified	Size
Synth Design (synth_design)		
Vivado Synthesis Report	7/10/12 9:29 AM	423.3 KB
Utilization Report	7/10/12 9:29 AM	7.0 KB
Place Design (place_design)		
Place and Route Log	7/11/12 6:50 PM	21.7 KB
IO Report	7/11/12 6:44 PM	146.4 KB
Clock Utilization Report	7/11/12 6:44 PM	20.9 KB
Utilization Report	7/11/12 6:44 PM	9.3 KB
Control Sets Report	7/11/12 6:44 PM	138.5 KB
Route Design (route_design)		
WebTalk Report		
DRC Report	7/11/12 6:49 PM	29.9 KB
Power Report	7/11/12 6:50 PM	30.9 KB
Route Status Report	7/11/12 6:50 PM	0.6 KB
Timing Summary Report	7/11/12 6:50 PM	470.5 KB
Write Bitstream (write_bitstream)		
WebTalk Report		

Figure 1-25: Reports Window

The reports available from the Reports window contain information related to the run. The selected report opens in text form in the Vivado IDE. See [Figure 1-26, Control Sets Report](#).



```

1 Control Set Information:
2 Number of unique control sets: 550
3 Minimum Number of register sites lost to control set restrictions: 1274
4 -----
5 | Clock Signal                               | Enable Signal
6 -----
7 | mgtEngine/gt_usrclk_source/GT4_TXUSRCLK_OUT | mgtEngine/gt5_frame_check/n_14970_373
8 | mgtEngine/gt_usrclk_source/GT4_TXUSRCLK_OUT | mgtEngine/gt4_frame_check/n_14970_372
9 | mgtEngine/gt_usrclk_source/GT6_TXUSRCLK_OUT | mgtEngine/gt7_frame_check/n_14970_375
10 | mgtEngine/gt_usrclk_source/GT6_TXUSRCLK_OUT | mgtEngine/gt6_frame_check/n_14970_374
11 | mgtEngine/gt_usrclk_source/GT0_TXUSRCLK_OUT | mgtEngine/gt0_frame_check/n_14970_368
12 | mgtEngine/gt_usrclk_source/GT2_TXUSRCLK_OUT | mgtEngine/gt2_frame_check/n_14970_370
13 | mgtEngine/gt_usrclk_source/GT0_TXUSRCLK_OUT | mgtEngine/gt1_frame_check/n_14970_369
14 | clkgen/cpuClk_o                            | cpuEngine/or1200_cpu/or1200_mult_mac/072
15 | clkgen/phyClk0_o                           | usbEngine0/u0/u0/bit_stuff_on
16 | clkgen/phyClk0_o                           | usbEngine0/u0/u0/suspend_set
17 | clkgen/fftClk_o                             |
18 | mgtEngine/gt_usrclk_source/GT2_TXUSRCLK_OUT | mgtEngine/gt3_frame_check/n_14970_371
19 | clkgen/phyClk1_o                           | usbEngine1/u0/u0/bit_stuff_on
20 | clkgen/fftClk_o                             | fftEngine/fftInst/egressLoop[7].egressFifo/buffer_fifo/SR[0]
21 | clkgen/phyClk1_o                           | usbEngine1/u0/u0/suspend_set
22 | mgtEngine/gt_usrclk_source/GT4_TXUSRCLK_OUT |
23 | mgtEngine/gt_usrclk_source/GT4_TXUSRCLK_OUT |
24 | mgtEngine/gt_usrclk_source/GT4_TXUSRCLK_OUT |
25 | clkgen/wbClk_o                              | wbArbEngine/s15/next
26 | clkgen/wbClk_o                              | cpuEngine/iwb_biu/n_14970_15354
27 | mgtEngine/gt_usrclk_source/GT6_TXUSRCLK_OUT |

```

Figure 1-26: Control Sets Report

Cross Probing from Reports

In both Project Mode and Non-Project Mode, the Vivado IDE supports cross-probing between reports and the associated design data in different windows (for example, the Device window).

- You generate the report using a menu command or Tcl command.
- Text reports do not support cross-probing.

For example, the Reports window includes a text-based Timing Summary Report under Route Design. See [Figure 1-25, Reports Window](#).

When analyzing timing, it is helpful to see the design data associated with critical paths, including placement and routing resources in the Device window.

To regenerate the report in the Vivado IDE, select **Tools > Timing > Report Timing Summary**. The resulting report allows you to cross-probe among the various views of the design.

Cross-Probing Between Timing Report and Device Window Example

Figure 1-27, [Cross-Probing Between Timing Report and Device Window](#), shows an example of cross-probing between the Timing Summary report and the Device window. The following steps took place in this Non-Project Mode example:

- A post-route design checkpoint was opened in the Vivado IDE.
- The Timing Summary report was generated and opened using `report_timing_summary -name`.
- The Routing Resources were enabled in the Device window.
- When the timing path was selected in the Timing Summary report, the path was automatically cross-probed in the Device window. See the following figure.

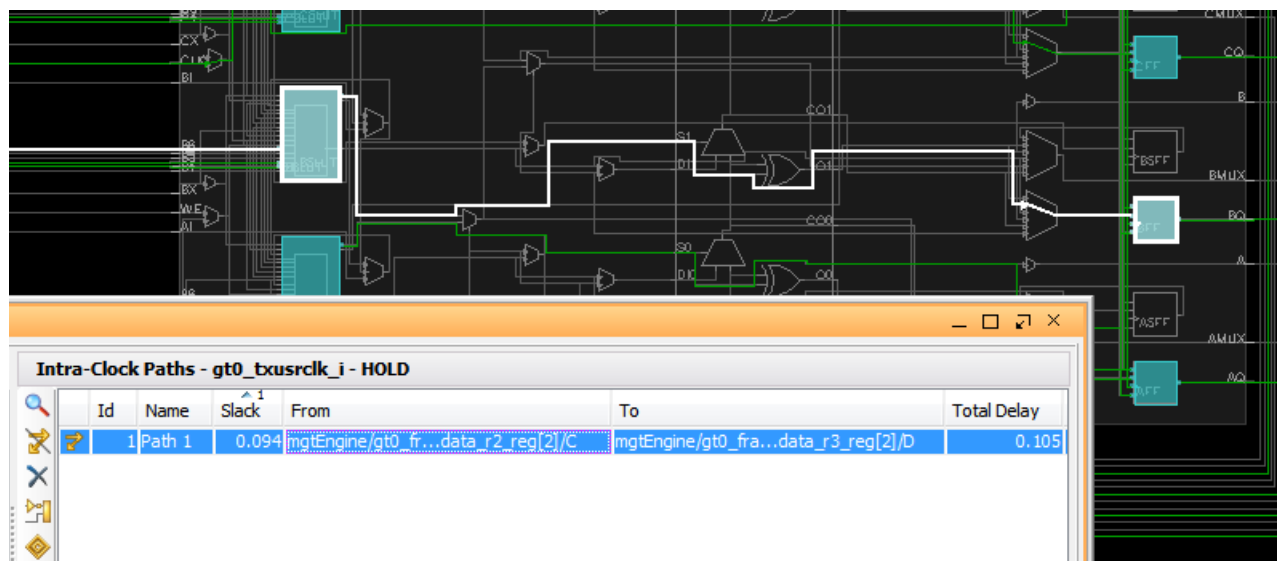


Figure 1-27: Cross-Probing Between Timing Report and Device Window

For more information on analyzing reports and strategies for design closure, see the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques (UG906)* [Ref 8].

Implementation Commands

About Implementation Commands

The Xilinx® Vivado™ Integrated Design Environment (IDE) includes many features to manage and simplify the implementation process for project-based designs. These features include the ability to step manually through the implementation process.

For more information, see [Running Implementation in Project Mode, page 20](#), in [Chapter 1, Vivado Implementation Process](#).

Non-Project based designs must be manually taken through each step of the implementation process using Tcl commands or Tcl scripts.

Note: For more information about Tcl commands, see the *Vivado Design Suite Tcl Command Reference Guide (UG835)* [[Ref 12](#)], or type `<command> -help`.

For more information, see [Running Implementation in Non-Project Mode, page 17](#), in [Chapter 1, Vivado Implementation Process](#).

Implementation Sub-Processes

Putting a design through the Vivado implementation process, whether in Project Mode or Non-Project Mode, consists of several sub-processes:

- **Open Synthesized Design:** Combines the netlist, the design constraints, and Xilinx target part data, to build the in-memory design to drive implementation.
- **Opt Design:** Optimizes the logical design and fits it onto the target Xilinx FPGA device.
- **Power Opt Design:** Optimizes elements of the design to reduce power demands of the implemented FPGA device.

Note: This step is optional.

- **Place Design:** Places the design onto the target Xilinx device.
- **Phys Opt Design:** Optimizes timing on the negative-slack paths of a design using various physical optimization techniques.

Note: This step is optional.

- **Route Design:** Routes the design onto the target Xilinx device.
- **Write Bitstream:** Generates a bitstream for Xilinx device configuration.

Note: Although not technically part of an implementation run, Write Bitstream is available as a separate step.

To provide a better understanding of the individual steps in the implementation process, the details of each step, and the associated Tcl commands, are documented in this chapter.

For a complete description of the Tcl reporting commands and their options, see the *Vivado Design Suite Tcl Command Reference Guide (UG835)*.

Opening the Synthesized Design

The first steps in implementation are to read the netlist from the synthesized design into memory and apply design constraints. You can open the synthesized design in various ways, depending on the flow used.

Creating the In-Memory Design

To create the in-memory design, the Vivado Design Suite uses the following process to combine the netlist files, constraint files, and the target part information:

1. Assembles the netlist.

The netlist is assembled from multiple sources if needed. Designs can consist of a mix of structural Verilog, EDIF, and NGC.

2. Transforms legacy netlist primitives to the currently supported subset of Unisim primitives.



TIP: Use `report_transformed_primitives` to generate a list of transformed cells.

3. Builds shapes.

- The Vivado tools create implicit shapes of cells based on their connectivity or placement constraints to simplify placement.

- Examples of implicit shapes include:
 - A relatively placed macro (RPM).

Note: RPMs are placed as a group rather than as individual cells.
 - A long carry chain that needs to be placed in multiple slices.

Note: The CARRY4 elements making up the carry chains must belong to a single shape to ensure downstream placement aligns it into vertical slices.

Tcl Commands

The following Tcl commands can be used to read the synthesized design into memory, depending on the source files in the design, and the state of the design:

- [synth_design](#)
- [read_checkpoint](#)
- [open_run](#)
- [link_design](#)

Table 2-1: Modes in Which Tcl Commands Can Be Used

Command	Project Mode	Non-Project Mode
synth_design	X	X
read_checkpoint		X
open_run	X	
link_design	X	X

synth_design

The **synth_design** command can be used in both Project Mode and Non-Project Mode. It runs Vivado synthesis on RTL sources with the specified options, and reads the design into memory after synthesis.

```
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>] [-mode <arg>] [-fsm_extraction <arg>]
  [-keep_equivalent_registers] [-quiet] [-verbose]
```

synth_design Example Script

The following is an excerpt from the `create_bft_batch.tcl` script found in the `examples/Vivado_Tutorials` directory of the software installation.

```
# 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
```

For more information on using the **synth_design** example script, see the *Vivado Design Suite Tutorial: Design Flows Overview (UG888)*.

The **synth_design** example script reads VHDL and Verilog files; reads a constraint file; and synthesizes the design on the specified part. The design is opened by the Vivado tool into memory when **synth_design** completes. A design checkpoint is written after completing synthesis.

For a complete description of the Tcl commands and their options, see the *Vivado Design Suite Tcl Command Reference Guide (UG835)*.

read_checkpoint

The **read_checkpoint** command can be used in Non-Project Mode only. It opens a post-synthesis design checkpoint for use in a non-project based design.

read_checkpoint Syntax

```
read_checkpoint [-part <arg>] [-quiet] [-verbose] <file>
```

read_checkpoint Example Script

```
# Read the specified design checkpoint and create an in-memory design.
read_checkpoint C:/Data/post_synth.dcp
```

The **read_checkpoint** example script opens the post synthesis design checkpoint file.

open_run

The **open_run** command opens a previously completed synthesis or implementation run, then loads the in-memory design of the Vivado tool.



IMPORTANT: The **open_run** command works in Project Mode only. Design runs are not supported in Non-Project Mode.

Use **open_run** before implementation on an RTL design in order to open a previously completed Vivado synthesis or XST run, then load the synthesized netlist into memory.



TIP: Because the in-memory design is updated automatically, you do not need to use **open_run** after **synth_design**. You need to use **open_run** only to open a previously completed synthesis run from an earlier design session.

The **open_run** command is for use with RTL designs only. To open a netlist-based design, use **link_design**.

open_run Syntax

```
open_run [-name <arg>] [-quiet] [-verbose] <run>
```

open_run Example Script

```
# Open named design from completed synthesis run
open_run -name synth_1 synth_1
```

The **open_run** example script opens a design (`synth_1`) into the Vivado tool memory from the completed synthesis run (also named `synth_1`).

If you use **open_run** while a design is already in memory, the Vivado tool prompts you to save any changes to the current design before opening the new design.

link_design

The **link_design** command creates an in-memory design from netlist sources (such as from a third-party synthesis tool), and links the netlists and design constraints with the target part.



TIP: The **link_design** command supports both Project Mode and Non-Project Mode to create the netlist design.

link_design Syntax

```
link_design [-name <arg>] [-part <arg>] [-constrset <arg>] [-top <arg>]
            [-mode <arg>] [-quiet] [-verbose]
```

link_design Example Script

```
# Open named design from netlist sources.
link_design -name netDriven -constrset constrs_1 -part xc7k325tfbg900-1
```

If you use **link_design** while a design is already in memory, the Vivado tool prompts you to save any changes to the current design before opening the new design.



RECOMMENDED: *After creating the in-memory synthesized design in the Vivado tool, review Errors and Critical Warnings for missing or incorrect constraints. After the design is successfully created, you can begin running analysis, generating reports, applying new constraints, or running implementation.*

Immediately after opening the in-memory synthesized design, run **report_timing_summary** to check timing constraints. This ensures that the design goals are complete and reasonable.

Logic Optimization

Logic optimization ensures the most efficient logic design before attempting placement. It performs a netlist connectivity check to warn of potential design problems such as nets with multiple drivers and un-driven inputs. Logic optimization also performs Block RAM power optimization.

Available Logic Optimizations

The Vivado tools can perform the following logic optimizations on the in-memory design:

- [Retargeting \(Default\)](#)
- [Constant Propagation \(Default\)](#)
- [Sweep \(Default\)](#)
- [Block RAM Power Optimization \(Default\)](#)
- [Remap](#)
- [Resynth Area](#)



IMPORTANT: *Expressly specifying one optimization disables the other optimizations, unless they are also expressly specified.*

Retargeting (Default)

Retargeting replaces one cell type with another to ease optimization. Example: A MUXF7 replaced by a LUT3 can be combined with other LUTs. In addition, simple cells such as inverters are absorbed into downstream logic.

Constant Propagation (Default)

Constant Propagation propagates constant values through logic, which results in:

- **Eliminated logic**

Example: an AND with a constant 0 input

- **Reduced logic**

Example: A 3-input AND with a constant 1 input is reduced to a 2-input AND.

- **Redundant logic**

Example: A 2-input OR with a logic 0 input is reduced to a wire.

Sweep (Default)

Sweep removes cells that have no loads.

Block RAM Power Optimization (Default)

Block RAM Power Optimization enables power optimization on Block RAM cells including:

- Changing the WRITE_MODE on unread ports of true dual-port RAMs to NO_CHANGE.
- Applying intelligent clock gating to Block RAM outputs.

Remap

Remap combines multiple LUTs into a single LUT to reduce the depth of the logic.

Resynth Area

Resynth Area performs re-synthesis in area mode to reduce the number of LUTs.



IMPORTANT: Each use of logic optimization affects the in-memory design, not the synthesized design that was originally opened.

opt_design

The **opt_design** command optimizes the current netlist. It also reads the in-memory design, optimizes it, and outputs the optimized design back into memory.

opt_design Syntax

```
opt_design [-retarget] [-propconst] [-sweep] [-bram_power_opt] [-remap]
[-resynth_area] [-directive <arg>] [-quiet] [-verbose]
```

opt_design Example Script

```
# Run logic optimization, save results in a checkpoint, report timing estimates
opt_design -retarget -propconst -sweep
write_checkpoint -force $outputDir/post_opt
report_timing_summary -file $outputDir/post_opt_timing_summary.rpt
```

The **opt_design** example script performs logic optimization on the in-memory design, rewriting it in the process. It also writes a design checkpoint after completing optimization, and generates a timing summary report and writes the report to the specified file.

Restrict Optimization to Listed Types

Use command line options to restrict optimization to one or more of the listed types. For example, use the following to skip block RAM optimization that is run by default:

```
opt_design -retarget -propconst -sweep
```

Using Directives

Directives provide different modes of behavior for the **opt_design** command. Only one directive may be specified at a time. The directive option is incompatible with other options. The following directives are available:

- **Explore**: Runs multiple passes of optimization.
- **ExploreArea**: Runs multiple passes of optimization with emphasis on reducing area.
- **AddRemap**: Runs the default logic optimization flow and includes LUT remapping to reduce logic levels.

Using the -verbose Option

To better analyze optimization results, use the **-verbose** option to see additional details of the logic affected by **opt_design** optimization.

The **-verbose** option is off by default due to the potential for a large volume of additional messages. Use the **-verbose** option if you believe it might be helpful.



IMPORTANT: The `opt_design` command operates on the in-memory design. If run multiple times, the subsequent run optimizes the results of the previous run.

Logic Optimization Constraints

1. The Vivado Design Suite respects the `DONT_TOUCH` and `MARK_DEBUG` properties during logic optimization. It does not optimize away nets with these properties.

For more information, see the *Vivado Design Suite User Guide: Synthesis (UG901)*.

`MARK_DEBUG` is placed on nets that are candidates for probing with the Vivado Logic Analyzer™ tool. A net with `MARK_DEBUG` is connected to a slice boundary to ensure it can be probed.

The `DONT_TOUCH` property is typically placed on leaf cells to prevent them from being optimized. `DONT_TOUCH` on a hierarchical cell preserves the cell boundary, but optimization may still occur within the cell.



IMPORTANT: `KEEP` and `KEEP_HIERARCHY` properties in designs migrated from the ISE Design Suite are automatically converted to `DONT_TOUCH`.

Power Optimization

Power optimization optimizes dynamic power using clock gating (optional). It can be used in both Project Mode and Non-Project Mode, and can be run after logic optimization or after placement to reduce power demand in the design. Power optimization includes Xilinx intelligent clock gating solutions that can reduce dynamic power in FPGA designs, but do not change the clocks or logic of the design.

For more information, see the *Vivado Design Suite User Guide: Power Analysis and Optimization (UG907)*.

Vivado Tool Power Analysis

The Vivado tool analyzes all portions of the design, including legacy and third-party IP blocks. It also identifies output logic from sourcing registers that does not contribute to the result for each clock cycle.

Using Clock Enables (CEs)

The Vivado power optimizer takes advantage of the abundant supply of Clock Enables (CEs) available in the logic of Xilinx 7 series FPGA devices. The tool creates fine-grain clock gating, or logic gating signals, that eliminate unnecessary switching activity in the register.

In addition, at the flip-flop level, CEs are actually gating the clock rather than selecting between the D input and feedback Q output of the flip-flop. This increases the performance of the CE input but also reduces clock power.

Intelligent Clock Gating

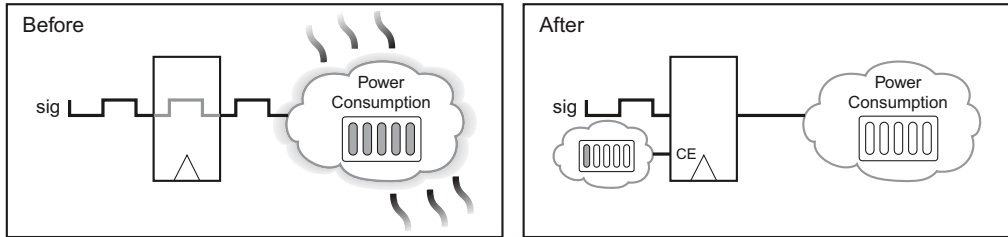


Figure 2-1: Intelligent Clock Gating

Intelligent clock gating also reduces power for dedicated block RAMs in either simple dual-port or true dual-port mode. See [Figure 2-2, Leveraging BRAM Enables](#).

These blocks include several enables:

- Array enable
- Write enable
- Output register clock enable

Most of the power savings comes from using the array enable. The Vivado power optimizer implements functionality to reduce power when no data is being written and when the output is not being used.

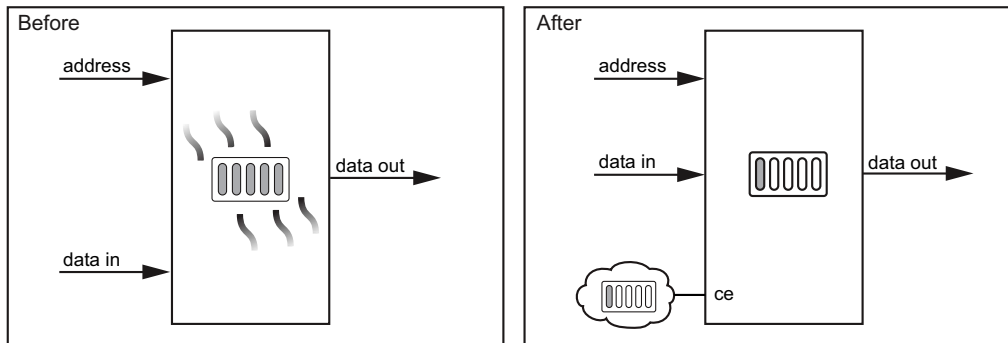


Figure 2-2: Leveraging BRAM Enables

power_opt_design

The **power_opt_design** command analyzes and optimizes the design. It analyzes and optimizes the entire design as a default. The command also performs intelligent clock gating to optimize power.

power_opt_design Syntax

```
power_opt_design [-quiet] [-verbose]
```

If you do not want to analyze and optimize the entire design, configure the optimizer with **set_power_opt**. This lets you specify the appropriate cell types or hierarchy to include or exclude in the optimization.

The syntax for **set_power_opt** is:

```
set_power_opt [-include_cells <args>] [-exclude_cells <args>] [-clocks <args>]  
              [-cell_types <args>] [-quiet] [-verbose]
```

Note: block RAM power optimization is skipped if it is run using **opt_design**.

Placement

The Vivado placer engine positions cells from the netlist onto specific sites in the target Xilinx device. Like the other implementation commands, the Vivado placer works from, and updates, the in-memory design.

Design Placement Optimization

The Vivado placer engine simultaneously optimizes the design placement for:

- **Timing slack:** Placement of cells in timing-critical paths is chosen to minimize negative slack.
- **Wirelength:** Overall placement is driven to minimize the overall wirelength of connections.
- **Congestion:** The Vivado placer monitors pin density and spreads cells to reduce potential routing congestion.

Design Rule Checks

Before starting placement, Vivado implementation runs Design Rule Checks (DRCs), including user-selected DRCs from **report_drc**, and built-in DRCs internal to the Vivado

placer engine. Internal DRCs report many issues, including memory Interface Generator (MIG) cells without LOC constraints, and I/O banks with conflicting IOSTANDARDS.

Clock and I/O Placement

After design rule checking, the Vivado placer places clock and I/O cells before placing other logic cells. Clock and I/O cells are placed concurrently because they are often related through complex placement rules specific to the targeted Xilinx device.

Placer Targets

The placer targets at this stage of placement are:

- I/O ports and logic
- Global and local clock buffers
- Clock management tiles (MMCMs and PLLs)
- Gigabit Transceiver (GT) cells

Placing Unfixed Logic

When placing unfixed logic during this stage of placement, the placer adheres to physical constraints, such as LOC properties and Pblock assignments. It also validates existing LOC constraints against the netlist connectivity and device sites. Certain IP (such as MIGs and GTs) are generated with device-specific placement constraints.



IMPORTANT: *Due to the device I/O architecture, a LOC property often constrains cells other than the cell to which LOC has been applied. A LOC on an input port also fixes the location of its related I/O buffer, IDELAY, and ILOGIC. Conflicting LOC constraints cannot be applied to individual cells in the input path. The same applies for outputs and GT-related cells.*

Clock Resources Placement Rules

Clock resources must follow the placement rules described in the *7 Series FPGAs Clocking Resources (UG472)* [Ref 11]. For example, an input that drives a global clock must be located at a clock-capable I/O site, and must be located in the same upper or lower half of the device. These clock placement rules are also validated against the logical netlist connectivity and device sites.

When Clock and I/O Placement Fails

If the Vivado placer fails to find a solution for the clock and I/O placement, the placer reports the placement rules that were violated, and briefly describes the affected cells. In some cases, the Vivado placer provisionally places cells at sites, and attempts to place other cells as it tries to solve the placement problem. The provisional placements often pinpoint

the source of clock and I/O placement failure. Manually placing a cell that failed provisional placement may help placement converge.



TIP: Use **place_ports** to run the clock and I/O placement step first. Then run **place_design**. If port placement fails, the placement is saved to memory to allow failure analysis. For more information, run **place_ports -help** from the Vivado Tcl command prompt.

Global Placement, Detailed Placement, and Packing and Legalization

After Clock and I/O placement, the remaining placement phases consist of:

- Global placement
- Detailed placement
- Packing and legalization

After placement, an estimated timing summary is output to the log file:

```
Phase 12 Placer Reporting  
INFO: [Place-100] Post Placement Timing Summary | WNS=-0.08836 | TNS=-1.479 |
```

where

- WNS = Worst Negative Slack
- TNS = Total Negative Slack



RECOMMENDED: Run **report_timing** after placement to check the critical paths. Paths with very large negative slack may need manual placement, further constraining, or logic restructuring to achieve timing closure.

place_design

The **place_design** command automatically places ports and cells. Like the other implementation commands, **place_design** is re-entrant or incremental in nature. For a partially placed design, the Vivado placer uses the existing placement as the starting point, instead of starting from scratch.

place_design Syntax

```
place_design [-directive <arg>] [-no_timing_driven] [-quiet] [-verbose]
```

place_design Example Script

```
# Run placement, save results to checkpoint, report timing estimates
place_design
write_checkpoint -force $outputDir/post_place
report_timing_summary -file $outputDir/post_place_timing_summary.rpt
```

The **place_design** example script places the in-memory design. It then writes a design checkpoint after completing placement, generates a timing summary report, and writes the report to the specified file.

Using Directives

Directives provide different modes of behavior for the **place_design** command. Only one directive may be specified at a time. The directive option is incompatible with other options.

Placer Directives

Because placement typically has the greatest impact on overall design performance, the Placer has the most directives of all commands. [Table 2-2, Directive Guidelines](#), shows which directives may benefit which types of designs.

Table 2-2: Directive Guidelines

Directive	Designs Benefitted
Block Placement	Designs with many Block RAM, DSP blocks, or both
NetDelay	Designs that anticipate many long-distance net connections and nets that fan out to many different modules
SpreadLogic	Designs with very high connectivity that tend to create congestion
ExtraPostPlacement Opt	All design types
SSI	SSI designs that may benefit from different styles of partitioning to relieve congestion or improve timing.

Available Directives

- **Explore:** Higher placer effort in detail placement and post-placement optimization.
- **WLDriVenBlockPlacemEnt:** Wirelength-driven placement of RAM and DSP blocks. Override timing-driven placement by directing the Placer to minimize the distance of connections to and from blocks.
- **LateBlockPlacemEnt:** Defers detailed placement of RAMB and DSP blocks to the final stages of placement. Normally blocks are committed to valid sites early in the placement process. Instead, the Placer uses coarse block placements that may not align with proper columns, then places blocks at valid sites during detail placement.

- **ExtraNetDelay_high:** Increases estimated delay of high fanout and long-distance nets. Three levels of pessimism are supported: high, medium, and low. **ExtraNetDelay_high** applies the highest level of pessimism.
- **ExtraNetDelay_medium:** Increases estimated delay of high fanout and long-distance nets. Three levels of pessimism are supported: high, medium, and low. **ExtraNetDelay_medium** applies the default level of pessimism.
- **ExtraNetDelay_low:** Increases estimated delay of high fanout and long-distance nets. Three levels of pessimism are supported: high, medium, and low. **ExtraNetDelay_low** applies the lowest level of pessimism.
- **SpreadLogic_high:** Spreads logic throughout the device. Three levels are supported: high, medium, and low. **SpreadLogic_high** achieves the highest level of spreading.
- **SpreadLogic_medium:** Spreads logic throughout the device. Three levels are supported: high, medium, and low. **SpreadLogic_medium** achieves a nominal level of spreading.
- **SpreadLogic_low:** Spreads logic throughout the device. Three levels are supported: high, medium, and low. **SpreadLogic_low** achieves a minimal level of spreading.
- **ExtraPostPlacementOpt:** Higher placer effort in post-placement optimization.
- **SSI_ExtraTimingOpt:** Use an alternate algorithm for timing-driven partitioning across SLRs.
- **SSI_SpreadSLLs:** Partition across SLRs and allocate extra area for regions of higher connectivity.
- **SSI_BalanceSLLs:** Partition across SLRs while attempting to balance SLLs between SLRs.
- **SSI_BalanceSLRs:** Partition across SLRs to balance number of cells between SLRs.
- **SSI_HighUtilSLRs:** Force the placer to attempt to place logic closer together in each SLR.
- **RuntimeOptimized:** Run fewest iterations, trade higher design performance for faster runtime.
- **Quick:** Absolute, fastest runtime, non-timing-driven, performs the minimum required for a legal design.
- **Default:** Run **place_design** with default settings.



TIP: Use the **-directive** option to explore different placement options for your design..

Using the `-no_timing_driven` Option

The `-no_timing_driven` option disables the default timing driven placement algorithm. This results in a faster placement based on wire lengths, but ignores any timing constraints during the placement process.

Using the `-verbose` Option

To better analyze placement results, use the `-verbose` option to see additional details of the cell and I/O placement by the `place_design` command.

The `-verbose` option is off by default due to the potential for a large volume of additional messages. Use the `-verbose` option if you believe it might be helpful.

Physical Optimization

Physical optimization optionally performs timing-driven optimization on the negative-slack paths of a design. Optimizations involve replication, retiming, hold fixing, and placement improvement. Physical optimization automatically performs all necessary netlist and placement changes

Available Physical Optimizations

The Vivado tools perform the following physical optimizations on the in-memory design:

- [High-Fanout Optimization \(Default\)](#)
- [Placement-Based Optimization \(Default\)](#)
- [Rewire \(Default\)](#)
- [Critical-Cell Optimization \(Default\)](#)
- [DSP Register Optimization \(Default\)](#)
- [BRAM Register Optimization \(Default\)](#)
- [Hold-Fixing](#)
- [Retiming](#)
- [Forced Net Replication](#)



IMPORTANT: *Expressly specifying one optimization disables the other optimizations, unless they are also expressly specified.*

High-Fanout Optimization (Default)

High-Fanout Optimization works as follows:

1. High fanout nets, with negative slack within a percentage of the WNS, are considered for replication.
2. Loads are clustered based on proximity, and drivers are replicated and placed for each load cluster.
3. Timing is re-analyzed, and logical changes are committed if timing is improved.
4. After replication, the design is checked again for high fanout nets to replicate. If high fanout nets still exist, the replication process continues until there are no high fanout nets to optimize.

Placement-Based Optimization (Default)

Placement-Based Optimization optimizes placement on the critical path by re-placing all the cells in the critical path to reduce wire delays.

Rewire (Default)

Rewire optimization the critical path by swapping connections on LUTs to reduce the number of logic levels for critical signals. LUT equations are modified to maintain design functionality.

Critical-Cell Optimization (Default)

Critical-Cell Optimization replicates cells in failing paths. If the loads on a specific cell are placed far apart, the cell may be replicated with new drivers placed closer to load clusters. High fanout is not a requirement for this optimization to occur, but the path must fail timing with slack within a percentage of the worst negative slack.

DSP Register Optimization (Default)

DSP Register Optimization can move registers out of the DSP cell into the logic array or from logic to DSP cells if it improves the delay on the critical path.

BRAM Register Optimization (Default)

BRAM Register Optimization can move registers out of the BRAM cell into the logic array or from logic to BRAM cells if it improves the delay on the critical path.

Hold-Fixing

Hold-Fixing attempts to improve slack of high hold violators by increasing delay on the hold critical path.

Retiming

Retiming improves the delay on the critical path by moving registers across combinational logic.

Forced Net Replication

Forced Net Replication forces the net drivers to be replicated, regardless of timing slack. Replication is based on load placements and requires manual analysis to determine if replication is sufficient. If further replication is required, nets can be replicated repeatedly by successive commands. Although timing is ignored, the net must be in a timing-constrained path to trigger the replication.

Physical Optimization Reports

Physical Optimization reports each net processed for optimization, and a summary of the optimization performed (if any).



TIP: *Replicated objects are named by appending `_replica` to the original object name, followed by the replicated object count.*

phys_opt_design



IMPORTANT: *The `phys_opt_design` command optionally performs physical optimizations such as timing-driven replication of high fanout nets to improve timing results. Running `phys_opt_design` on a routed design is not supported.*

phys_opt_design Syntax

```
phys_opt_design [-fanout_opt] [-placement_opt] [-rewire] [-critical_cell_opt]
[-dsp_register_opt] [-bram_register_opt] [-hold_fix] [-retime]
[-force_replication_on_nets <args>] [-directive <arg>] [-quiet] [-verbose]
```

phys_opt_design Example Script

```
# Run physical optimization, save results to checkpoint, report timing estimates
phys_opt_design
write_checkpoint -force $outputDir/post_phys_opt.dcp
report_timing_summary -file $outputDir/post_phys_opt_timing_summary.rpt
```

The **phys_opt_design** example script performs physical optimization on in-memory design, rewriting it in the process. It then writes a design checkpoint after completing physical optimization, and generates a timing summary report and writes the report to the specified file.

Using Directives

Directives provide different modes of behavior for the **phys_opt_design** command. Only one directive may be specified at a time and the directive option is incompatible with other options. The following directives are available:

- **Explore**: Run different algorithms in multiple passes of optimization, including hold violation fixing and replication for very high fanout nets.
- **AggressiveExplore**: Similar to Explore but with different optimization algorithms and more aggressive goals.
- **AlternateReplication**: Use different algorithms for performing critical cell replication.
- **AggressiveFanoutOpt**: Uses different algorithms for fanout-related optimizations with more aggressive goals.
- **AlternateDelayModeling**: Performs all optimizations using alternate algorithms for estimating net delays.
- **AddRetime**: Performs the default **phys_opt_design** flow and adds register retiming.
- **Default**: Run **phys_opt_design** with default settings.

Using the -verbose Option

To better analyze physical optimization results, use the **-verbose** option to see additional details of the optimizations performed by the **phys_opt_design** command.

The **-verbose** option is off by default due to the potential for a large volume of additional messages. Use the **-verbose** option if you believe it might be helpful.



IMPORTANT: The **phys_opt_design** command operates on the in-memory design. If run twice, the second run optimizes the results of the first run.

Physical Optimization Constraints

The Vivado Design Suite respects the DONT_TOUCH and MARK_DEBUG properties during physical optimization. It does not perform physical optimization on nets or cells with these properties. Additionally, Pblock assignments are obeyed such that replicated logic inherits the Pblock assignments of the original logic. Timing exceptions are also copied from original to replicated cells.

For more information, see the *Vivado Design Suite User Guide: Synthesis (UG901)*.

MARK_DEBUG is placed on nets that are candidates for probing with the Vivado™ Logic Analyzer tool. A net with MARK_DEBUG is connected to a slice boundary to ensure it can be probed.

The DONT_TOUCH property is typically placed on leaf cells to prevent them from being optimized. DONT_TOUCH on a hierarchical cell preserves the cell boundary, but optimization may still occur within the cell.



IMPORTANT: *KEEP and KEEP_HIERARCHY properties in designs migrated from the ISE Design Suite are automatically converted to DONT_TOUCH.*

Routing

The Vivado router performs routing on the placed design, and performs optimization on the routed design to resolve hold time violations. It is timing driven by default, although this can be disabled.

Routing Modes

The router can be run in two modes:

- [Normal Mode](#)
- [Re-Entrant Mode](#)

Normal Mode

Normal Mode is the default. In Normal Mode, the Vivado router starts with a placed design, and attempts to route all nets.

The Vivado router can start with a placed design that is:

- Unrouted
- Partially routed
- Fully routed

The **route_design** command is incremental in nature. For a partially routed design, the Vivado router uses the existing routes as the starting point, instead of starting from scratch.

Re-Entrant Mode

Like other implementation commands, the router is *re-entrant*. Re-entrant means that the router continues routing a design with existing routes, instead of discarding them and starting from scratch.



RECOMMENDED: *Because routing initialization is costly in terms of runtime (especially for multiple re-entrant routing passes), use Re-Entrant Mode when you intend to run multiple routing passes for a sequence of operations.*

Without Re-Entrant Mode, the Vivado router exits and clears its memory after each routing operation, requiring the router to be initialized each time it is run. This can be time consuming for multiple routing steps.

In Re-Entrant Mode, the Vivado router keeps its data structures in memory in anticipation of subsequent routing operations. Subsequent commands such as unrouting and routing individual nets can be performed immediately.

Design Rule Checks

Before starting routing, the Vivado tool runs Design Rule Checks (DRC), including:

- User-selected DRCs from **report_drc**
- Built-in DRCs internal to the Vivado router engine

Routing Priorities

The Vivado Design Suite routes global resources first, such as:

- Clocks
- Resets
- I/O
- Other dedicated resources

This default priority is built into the Vivado router. The router then prioritizes data signals according to timing criticality.

Impact of Poor Timing Constraints

When using either pre-route flows or fixed routing constraints, keep in mind that often the Vivado router cannot optimally route some signals because poor timing constraints are giving the router an incorrect timing picture.

Common examples of poor timing constraints include:

- Cross-clock paths and multi-cycle paths in which hold timing causes route delay insertion
- Congested areas, which can be addressed by targeted fanout optimization in RTL synthesis or through physical optimization

Nets that are routed sub-optimally are often the result of incorrect timing constraints. Before you experiment with router settings, make sure that you have validated the constraints and the timing picture seen by the router. Validate timing and constraints by reviewing timing reports from the placed design before routing.



RECOMMENDED: *Clean up constraints (or consider RTL changes) rather than simply increasing routing resources.*

route_design

Route the nets in the current design to complete logic connections on the target part.

route_design Syntax

```
route_design [-unroute] [-re_entrant <arg>] [-nets <args>] [-physical_nets]
[-pin <arg>] [-directive <arg>] [-no_timing_driven] [-preserve] [-delay]
[-free_resource_mode] -max_delay <arg> -min_delay <arg>
[-quiet] [-verbose]
```

Using Directives

Directives provide different modes of behavior for the **route_design** command. Only one directive may be specified at a time. The directive option is incompatible with other options. The following directives are available:

- **Explore:** Allows the router to explore different critical path placements after an initial route.
- **NoTimingRelaxation:** Prevents the router from relaxing timing to complete routing. If the router has difficulty meeting timing, it will run longer to try to meet the original timing constraints.
- **MoreGlobalIterations:** Uses detailed timing analysis throughout all stages instead of just the final stages, and will run more global iterations even when timing improves only slightly.
- **HigherDelayCost:** Adjusts the router's internal cost functions to emphasize delay over iterations, allowing a tradeoff of runtime for better performance.
- **AdvancedSkewModeling:** Uses more accurate skew modeling throughout all routing stages which may improve design performance on higher-skew clock networks.

- **RuntimeOptimized:** Run fewest iterations, trade higher design performance for faster runtime.
- **Quick:** Absolute, fastest runtime, non-timing-driven, performs the minimum required for a legal design.
- **Default:** Run `route_design` with default settings.

Trading Runtime for Better Routing

The following directives are methods of trading runtime for potentially better routing results:

- NoTimingRelaxation
- MoreGlobalIterations
- HigherDelayCost
- AdvancedSkewModeling

Normal Mode Routing Example Script

```
# Route design, save results to checkpoint, report timing estimates
route_design
write_checkpoint -force $outputDir/post_route
report_timing_summary -file $outputDir/post_route_timing_summary.rpt
```

The **route_design** example script places the in-memory design, rewriting it in the process. It then writes a design checkpoint after completing routing, generates a timing summary report, and writes the report to the specified file.

Normal mode routing is performed as part of an implementation run, or by running **route_design** after **place_design** as part of a Tcl script.

After routing is complete, routing statistics summarize the routing resources used by resource type, and display a timing summary:

```
[Route-20] Post Routing Timing Summary | WNS=0.0585 | TNS=0 | WHS=0 | THS=0 |
```

where

- WNS = Worst Negative Slack
- TNS = Total Negative Slack
- WHS = Worst Hold Slack
- THS = Total Hold Slack

Re-Entrant Mode Routing Example Script One

```
# route a few critical nets
route_design -delay -nets [get_nets myPreRoutes*]
# Complete full route
route_design
```

Re-entrant mode is implicitly entered when using a re-entrant routing option, such as **-nets** or **-pin**.

Re-entrant mode is usually run interactively to address specific routing issues such as:

- Pre-routing critical nets and locking down resources before a full route.
- Manually unrouting non-critical nets to free up routing resources for more critical nets.

The first re-entrant route command initializes the router and routes essential nets, such as clocks. This allows the router to perform:

- Timing analysis
- Timing-driven routing
- Hold-fixing

After re-entrant mode is enabled, the results of iterative routing and unrouting reside in memory.

Exiting Re-Entrant Mode

To exit re-entrant mode run the full **route_design** command, or directly disable re-entrant routing:

```
route_design -re_entrant off
```

Re-Entrant Mode Routing Example Script Two

```
# Get the nets in the top 10 critical paths, assign to $preRoutes
% set preRoutes [get_nets -of [get_timing_paths -max_paths 10]]
% route_design -nets [get_nets $preRoutes] -delay

# Unroute all the nets in u0/u1, and route the critical nets first
route_design -unroute [get_nets u0/u1/*]
% route_design -delay -nets [get_nets $myCritNets]
route_design -effort_level high
```

The strategy in this example script is to:

- Identify the top ten critical paths using **get_timing_paths**.
- Create a list of the net objects (**\$preRoutes**) of those critical paths using **get_nets -of**.
- Route those nets first.

The script continues after routing the pre-route nets.

After **route_design** completes, the Vivado router unroutes all nets in cell **u0/u1**, then re-routes identified critical nets first (**myCritNets**). The general router finishes any remaining unrouted nets.

Table 2-3: **Commands Used During Routing for Design Analysis**

Command	Function
report_route_status	Reports route status for nets
report_timing	Performs path endpoint analysis

For a complete description of the Tcl reporting commands and their options, see the *Vivado Design Suite Tcl Command Reference Guide (UG835)*.

Modifying Routing and Logic

Introduction to Modifying Routing and Logic

The Xilinx® Vivado™ Design Suite provides several ways to modify routing and logic in the implementation flow. These methods allow you to precisely control routing and delays and make quick logic changes.

- [Modifying Routing](#)
 - [Modifying Logic](#)
-

Modifying Routing

The Device View allows you to modify the routing for your design. You can Unroute, Route, and Fix Routing on any individual net.

- **Unroute and Route:** Calls the router in re-entrant mode to perform the operation on the net. For more information, see [route_design, page 86](#), in [Chapter 2, Implementation Commands](#).
- **Fix Routing:** Deposits the route, marks it fixed in the route database, and fixes the LOC and BEL of the driver and the load of the net. You can also enter Assign Routing Mode, which allows you to manually route a net. For more information, see [Manual Routing, page 91](#).



TIP: All net commands are available from the context menu on a net.

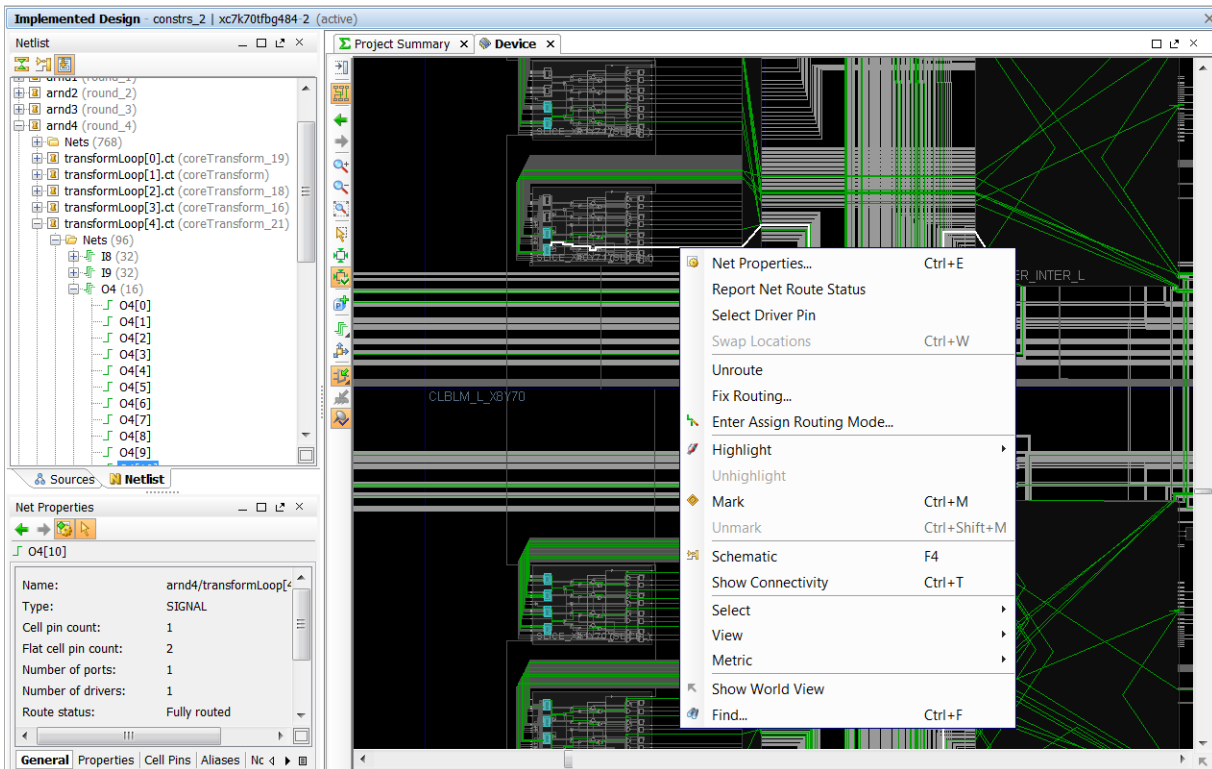


Figure 3-1: Modify Routing

Manual Routing

Manual routing allows you to select specific routing resources for your nets. This gives you complete control over the routing paths that a signal is going to take. Manual routing does not invoke **route_design**. Routes are directly updated in the route database.

You may want to use manual routing when you want to precisely control the delay for a net. For example, assume a source synchronous interface, in which you want to minimize routing delay variation to the capture registers in the device. To accomplish this, you can assign LOC and BEL constraints to the registers and I/Os, and then precisely control the route delay from the IOB to the register by manual routing the nets.

Manual routing requires detailed knowledge of a device's interconnect architecture. It is best used for a limited number of signals and for short connections.

Manual Routing Rules

Observe these rules during manual routing:

- The driver and the load require a LOC constraint and a BEL constraint.
- Branching is not allowed during manual routing, but you can implement branches by starting a new manual route from a branch point.

- LUT loads must have their pins locked.
- You must route to loads that are not already connected to a driver.
- Only complete connections are permitted. Antennas are not allowed.
- Overlap with existing unfixed routed nets is allowed. Run **route_design** after manual routing to resolve any conflicts due to overlapping nets.

Entering Assign Routing Mode

To enter Assign Routing Mode:

1. Open Device View.
2. Select the net that requires routing.
 - Unrouted nets are indicated by a red flyline.
 - Partially routed nets are highlighted in yellow.
3. Right click and select **Enter Assign Routing Mode**.

The Target Load Cell Pin window opens.

4. Optionally select a load cell pin to which you want to route.
5. Click **OK**.

You are now in Manual Routing Mode. A Routing Assignment window appears next to the Device View. See the following figure (Routing Assignment Window).

Routing Assignment Window

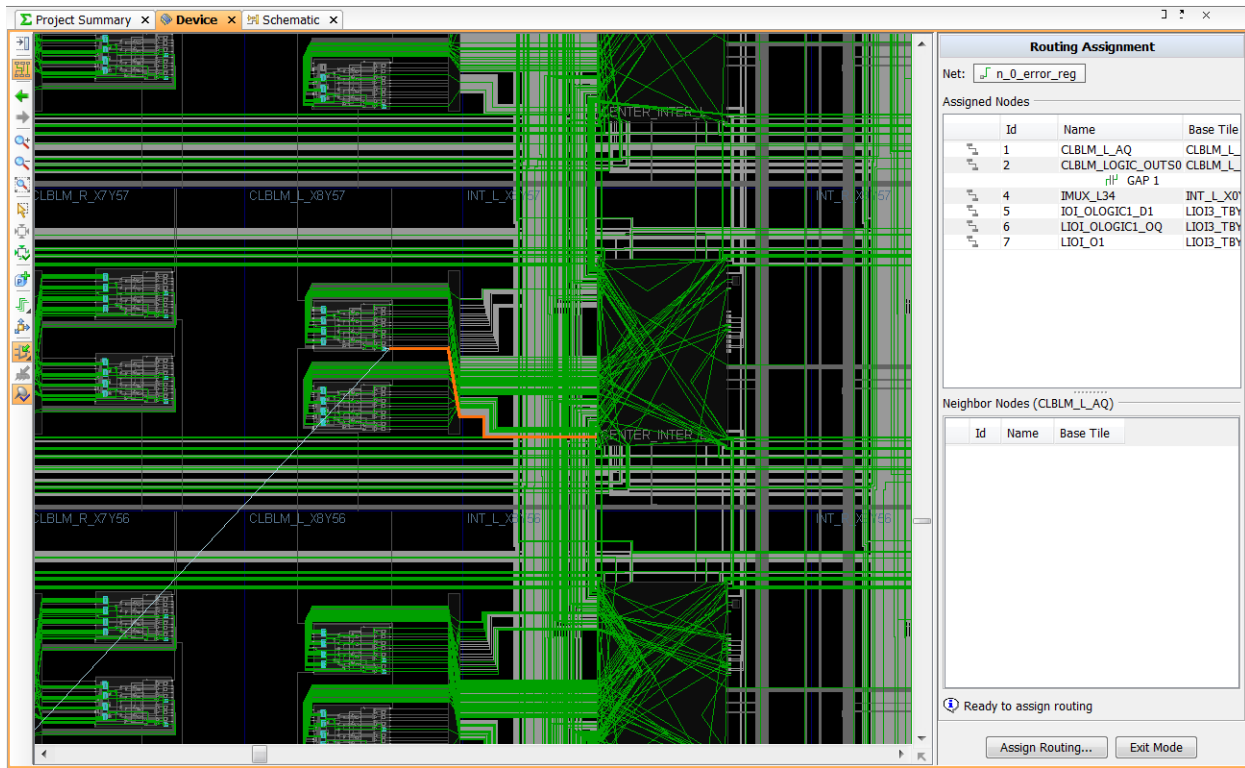


Figure 3-2: Routing Assignment Window

The routing assignment window is divided into two sections:

- Assigned Nodes
- Neighbor Nodes

The Assigned Nodes section shows the nodes that already have assigned routing. Each assigned node is displayed as a separate line item.

In the Device View, nodes with assigned routing are highlighted in orange. Any gaps between assigned nodes are shown in the Assigned nodes section as a GAP line item.

To assign the next routing segment, select an assigned node before or after a gap, or the last assigned node in the Assigned Nodes section.

This displays the allowed neighbor nodes in the Neighbor Nodes section. It also highlights the current selected nodes (in white) and the allowed neighbor nodes (white dotted) in the Device View. See the following figure (Assign Next Routing Segment).

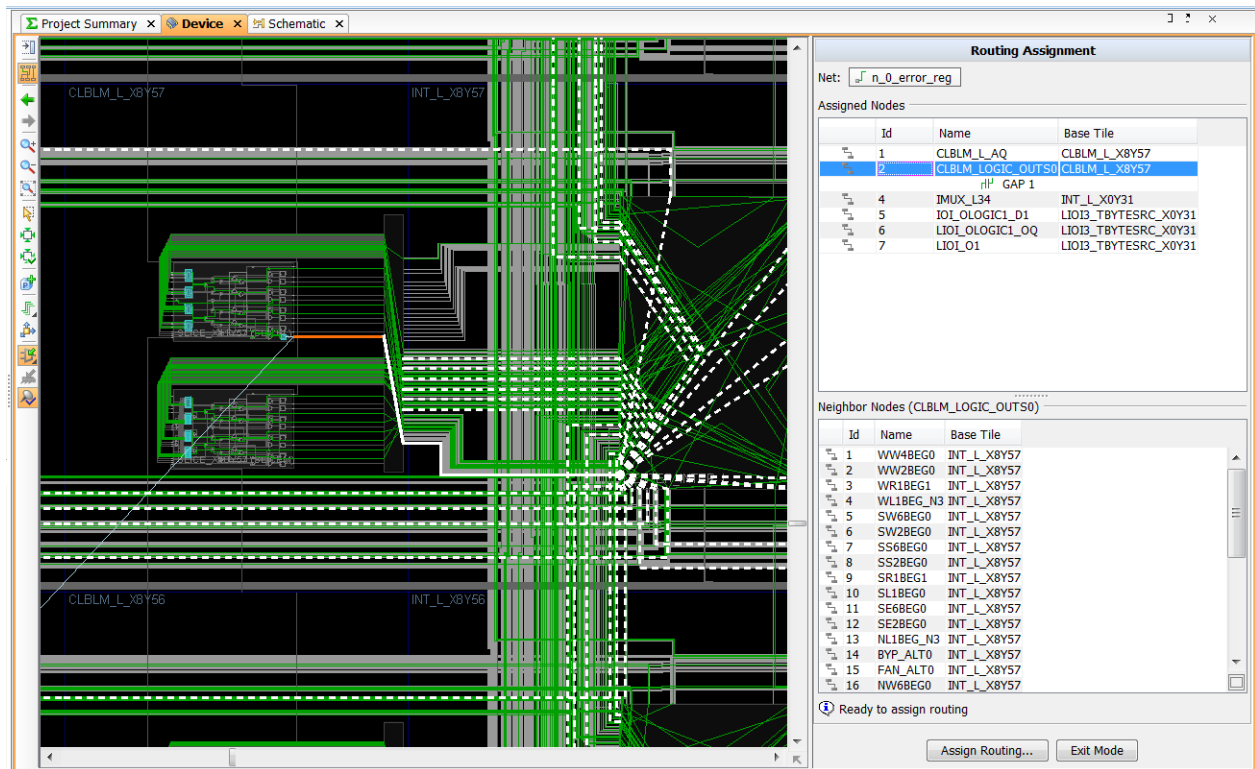


Figure 3-3: Assign Next Routing Segment

Assigning Routing Nodes

Once you have decided which Neighbor Node to assign for your next route segment, you can:

- Right click the node in the Neighbor Nodes section and select **Assign Node**.
- Double click the node in the Neighbor Nodes section.
- Single click the node in the Device View

Once you have assigned routing to a Neighbor Node, the node is displayed in the assigned nodes section and highlighted in orange in the Device View.

Assign nodes until you have reached the load, or until you are ready to assign routing with a gap.

Un-Assigning Routing Nodes

To un-assign nodes:

1. Go to the Assigned Nodes pane of the Routing Assignment window.
2. Select the nodes to be un-assigned.
3. Right-click and select **Remove**.

The nodes are removed from the assignment.

Exiting Assign Routing Mode

To finish the routing assignment and exit Assign Routing Mode:

1. Click the **Assign Routing** button in the Routing Assignment window.
2. Select the net in the Device View.
3. Right click and select **Assign Routing**.

The Assign Routing Window is displayed, allowing you to verify the assigned nodes before they are committed. See the following figure (Assign Routing Confirmation).

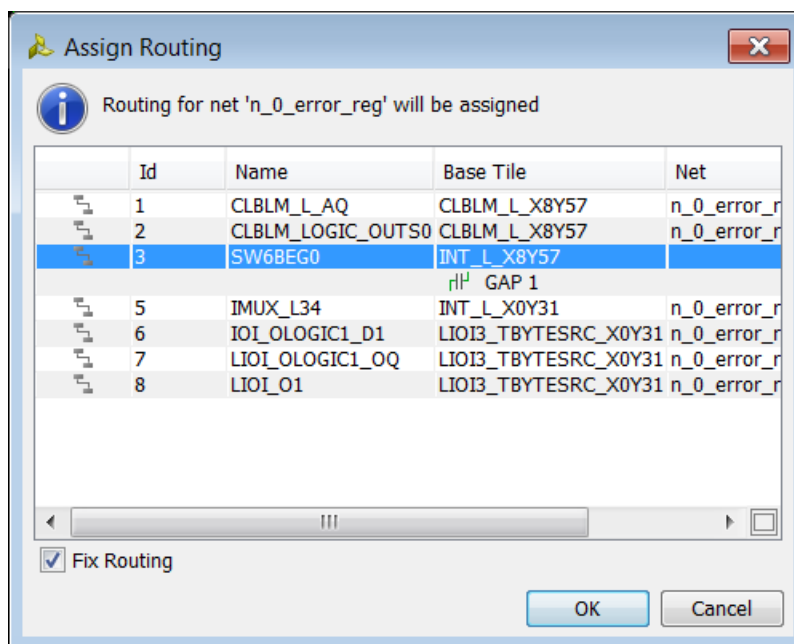


Figure 3-4: Assign Routing Confirmation

Canceling Out of Assign Routing Mode

If you are not ready to commit your routing assignments, you can cancel out of the Assign Routing Mode using one of the following methods:

- Click **Exit Mode** in the Routing Assignment window, or
- Right click in the Device View and select **Exit Assign Routing Mode**.

When the routes are committed, the driver and load BEL and LOC are also fixed.

Verifying Assigned Routes

- Assigned routes appear as dotted green lines in the Device View.
- Partially assigned routes appear as dotted yellow lines in the Device view.

The following figure (Assigned Partially Assigned Routing) shows an example of an assigned and partially assigned route.

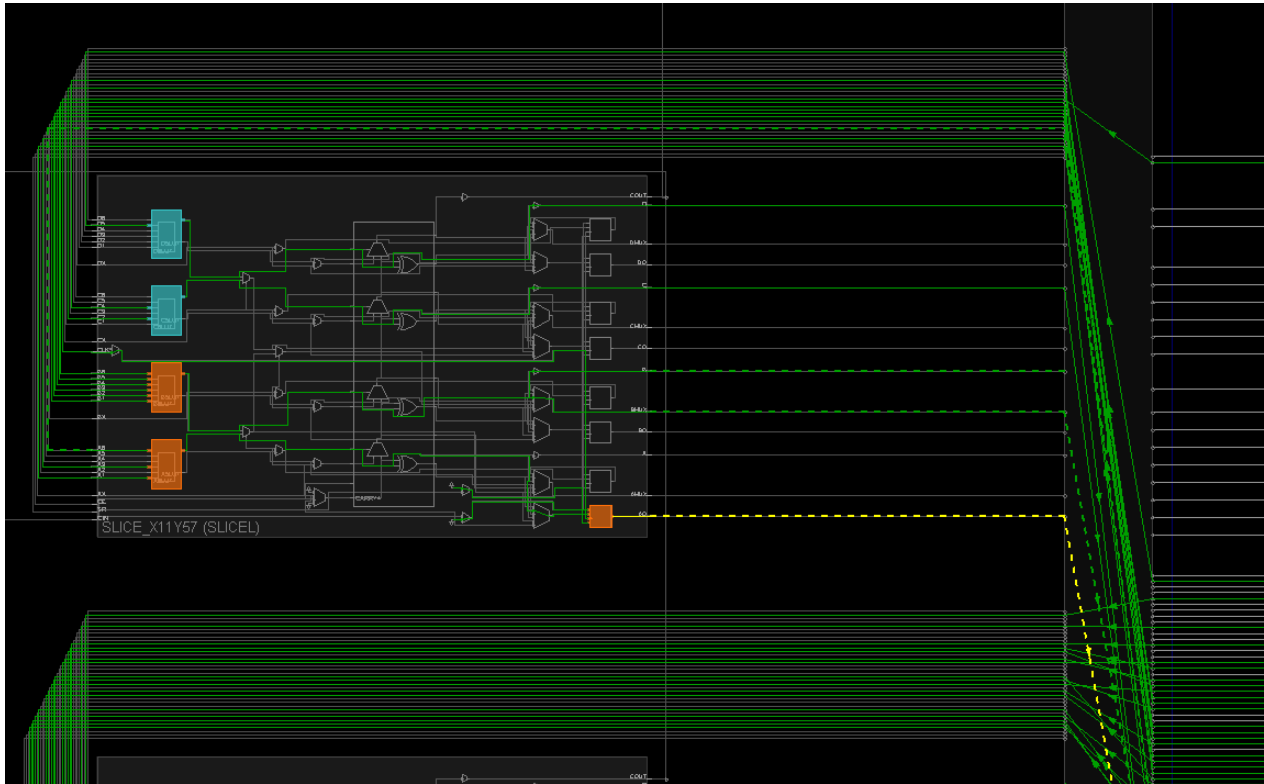


Figure 3-5: Assigned Partially Assigned Routing

Locking Cell Inputs on LUT Loads

You must ensure that the inputs of LUT loads to which you are routing are not being swapped with other inputs on those LUTs. To do so, lock the cell inputs of LUT loads as follows:

1. Open Device View.
2. Select the load LUT.
3. Right click and select **Lock Cell Input Pins**.

Branching

When assigning routing to a net with more than one load, you must route the net in the following steps:

1. Assign routing to one load following the steps shown in [Entering Assign Routing Mode](#), [page 92](#) above.
2. Assign routing to all the branches of the net.
3. The following figure (Assign Branching Route) shows an example of a net that has assigned routing to one load and requires routing to two additional loads.

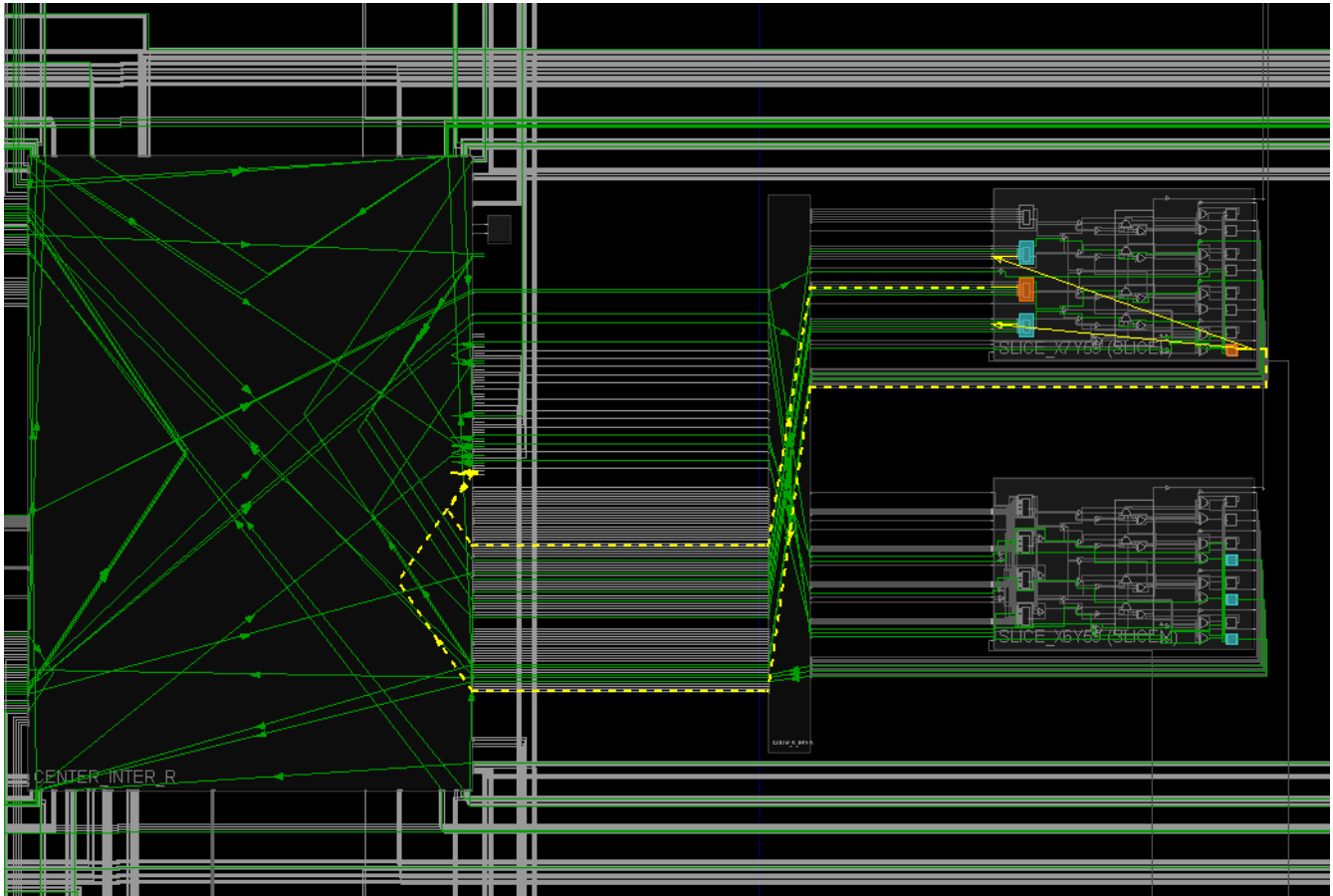


Figure 3-6: Assign Branching Route

Assigning Routing to a Branch

To assign routing to a branch:

1. Go to Device View.
2. Select the net to be routed.
3. Right click.
4. Select **Enter Assign Routing Mode**.

The Target Load Cell Pin window opens, showing all loads.

Note: The loads that already have assigned routing have a checkmark in the Routed column of the table.

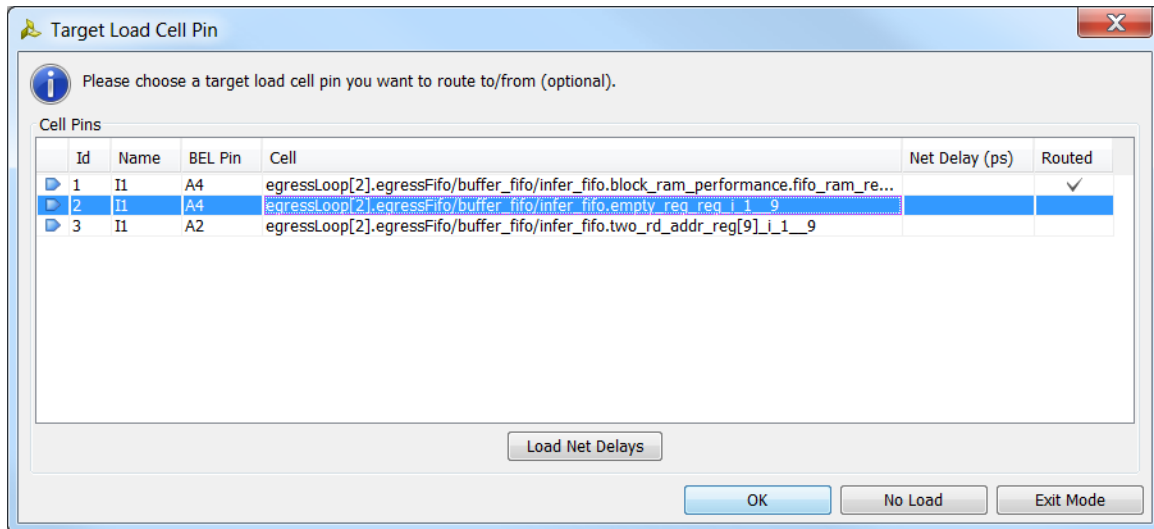


Figure 3-7: Target Load Cell Pin (Multiple Loads)

5. Select the load to which you want to route.
6. Click **OK**.

The Branch Start window opens.

See [Figure 3-8, Branch Start](#), page 100.

7. Select the node from which you want to branch off the route for your selected load.
8. Click **OK**.
9. Follow the steps shown in [Assigning Routing Nodes](#), page 94.

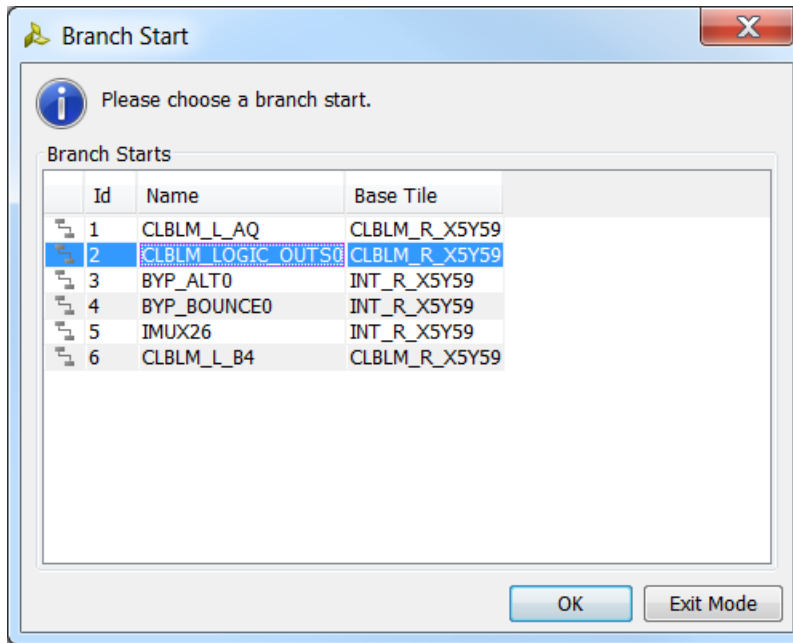


Figure 3-8: Branch Start

Directed Routing Constraints

Fixed route assignments are stored as Directed Routing Strings in the route database. In a Directed Routing String, branching is indicated by nested {curly braces}.

For example, consider the route described in [Figure 3-9, Branch Route Example, page 101](#). In that simplified picture of a route, the various elements are indicated as shown in the following table (Directed Routing Constraints).

Table 3-1: Directed Routing Constraints

Elements	Indicated By
Driver and Loads	Orange Rectangles
Nodes	Red lines
Switchboxes	Blue rectangles

A simplified version of a Directed Routing String for that route would appear as follows:

{ A B { D E L } C { F G H I M N } { O P Q } R J K L S }.

The route branches at B and C. The main trunk of this route is A B C R J K L S.

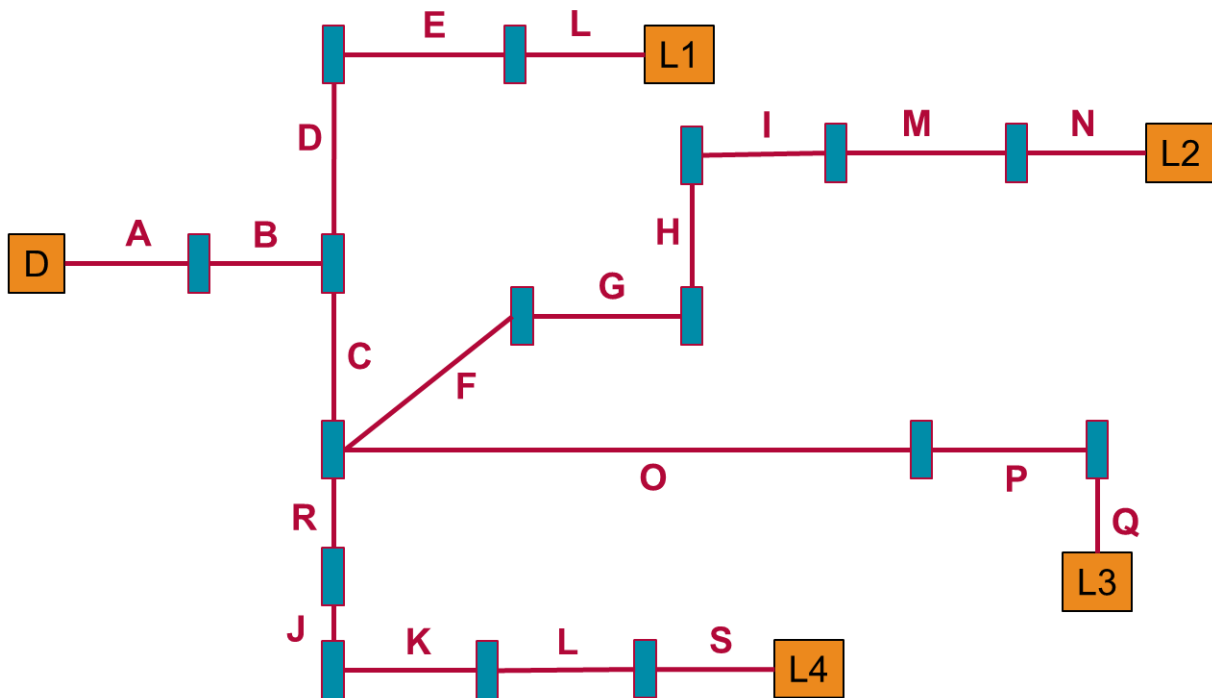


Figure 3-9: Branch Route Example

Modifying Logic

Properties on logical objects that are not Read Only can be modified after Implementation in the Vivado IDE graphical user interface as well as Tcl.

Note: For more information about Tcl commands, see the *Vivado Design Suite Tcl Command Reference Guide (UG835)* [Ref 12], or type **<command> -help**.

To modify a property on an object in Device View:

1. Select the object.
2. Modify the property value of the object in the Properties section of the Properties window.

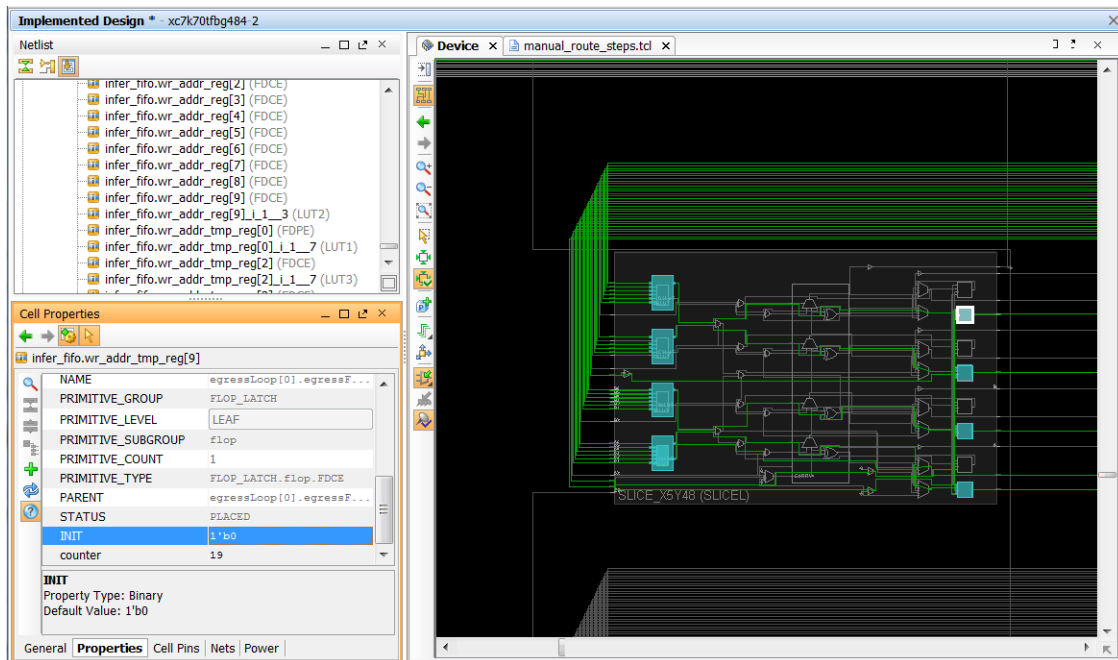


Figure 3-10: Property Modify

These properties can include everything from Block RAM INITs to the clock modifying properties on MMCMs. There is also a special dialog box to set or modify INIT on LUT objects. This dialog box allows you to specify the LUT equation and have the tool determine the appropriate INIT.

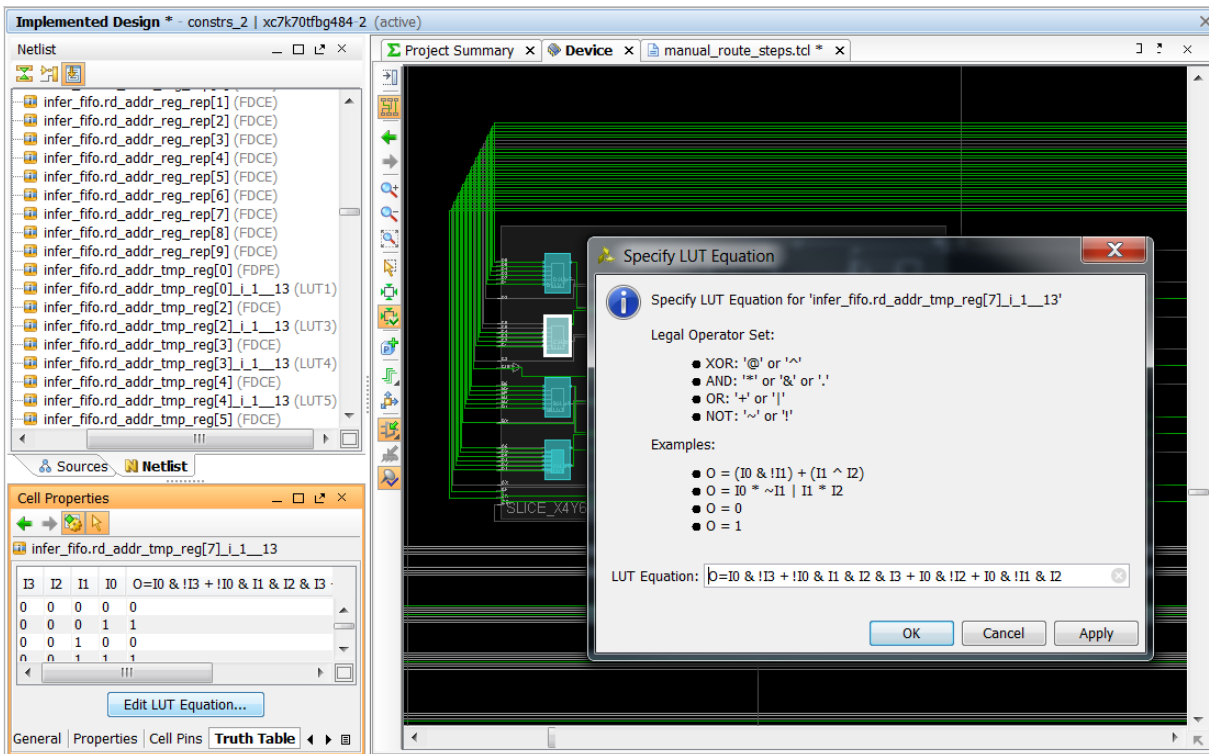


Figure 3-11: Equation Editor

Saving Modifications

To capture the changes to the design made in memory, write a checkpoint of the design.

Because the assignments are not back-annotated to the design, you must add the assignments to the XDC for them to impact the next run.

To save the constraints to your constraints file in Project Mode, select **File > Save Constraints**.

Using Remote Hosts

Launching Runs on Remote Linux Hosts

The Xilinx® Vivado™ Integrated Design Environment (IDE) supports simultaneous parallel execution of synthesis and implementation runs on multiple Linux hosts. This is accomplished in the application with simplified versions of more robust load-sharing software, such as Oracle Grid Engine and IBM® Platform™ LSF.

Linux is the only operating system supporting remote hosts because of:

- Superior security handling in Linux
- The lack of remote-shell capabilities on Microsoft Windows systems

Job submission algorithms are implemented using a “greedy,” round-robin style with Tcl pipes within Secure Shell (SSH), a service within the Linux operating system.



RECOMMENDED: *Before launching runs on multiple Linux hosts in the Vivado IDE, configure SSH so that the host does not require a password each time you launch a remote run.*

For instructions on configuring SSH, see [Setting Up SSH Key Agent Forward, page 108](#).

Launch Requirements

The requirements for launching synthesis and implementation runs on remote Linux hosts are:

- Vivado tool installation is assumed to be available from the login shell, which means that `$XILINX_VIVADO` and `$PATH` are configured correctly in your `.cshrc/` `.bashrc` setup scripts. If you can log into a remote machine and enter `vivado -help` without sourcing any other scripts, this flow should work.

If you do not have Vivado set up upon login (CSHRC or BASHRC), use the **Run pre-launch script** option, as described below, to define an environment setup script to be run prior to all jobs.

- Vivado IDE installation must be visible from the mounted file systems on remote machines. If the Vivado IDE installation is stored on a local disk on your own machine, it may not be visible from remote machines.
- Vivado IDE project files (.xpr) and directories (.data and .runs) must be visible from the mounted file systems on remote machines. If the design data is saved to a local disk, it may not be visible from remote machines.

Configuring Remote Hosts

To configure the Vivado IDE to run synthesis or implementation on a remote Linux host:

1. Select one of the following commands:
 - **Main Menu > Tools > Options > Remote Hosts**
 - **Flow Navigator > Synthesis > Launch Synthesis Runs > Configure Hosts**
 - **Flow Navigator > Implementation > Launch Implementation Runs > Configure Hosts**
 - **Launch Selected Runs Dialog Box > Configure Hosts**

See [Figure A-1, Configure Hosts from Launch Synthesis and Launch Implementation Runs](#).

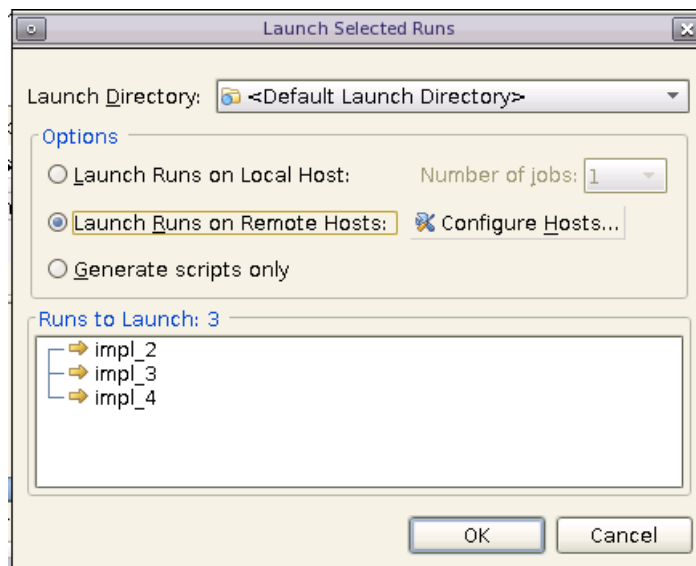


Figure A-1: Configure Hosts from Launch Synthesis and Launch Implementation Runs

The Vivado Options dialog box displays with the Remote Hosts section selected. The list of currently defined remote Linux hosts is displayed. See [Figure A-2, Configuring Remote Hosts](#).

2. Click **Add** to enter the names of additional remote servers.

3. Specify the number of processors the remote machine has available to run simultaneous processes using the **Jobs** field next to the host name. Individual runs require a separate processor.
4. Toggle **Enabled** to specify whether the server is available. Use this field when launching runs to specify which servers to use for selected runs.
5. Modify **Launch jobs with** to change the remote access command used when launching runs.

Note: This step is optional.

The default command is:

```
ssh -q -o -BatchMode=yes
```



IMPORTANT: Use caution when modifying this field. For example, removing **BatchMode =yes** might cause the remote process to hang because the Secure Shell incorrectly prompts for an interactive password.

6. Optionally, check **Run pre-launch script** and define a shell script to run before launching the runs. Use this option to run script to setup the host environment if you do not have Vivado IDE set up upon login.
7. Optionally, check **Run post-completion script** and define a custom script to run after the run completes, to move or copy the results for example.
8. Optionally, check **Send email to** and enter an Email address to send a notification when the runs complete. You can have notifications sent **After each Job**, or **After all jobs**.
9. Click **OK** to accept the Remote Host configuration settings.

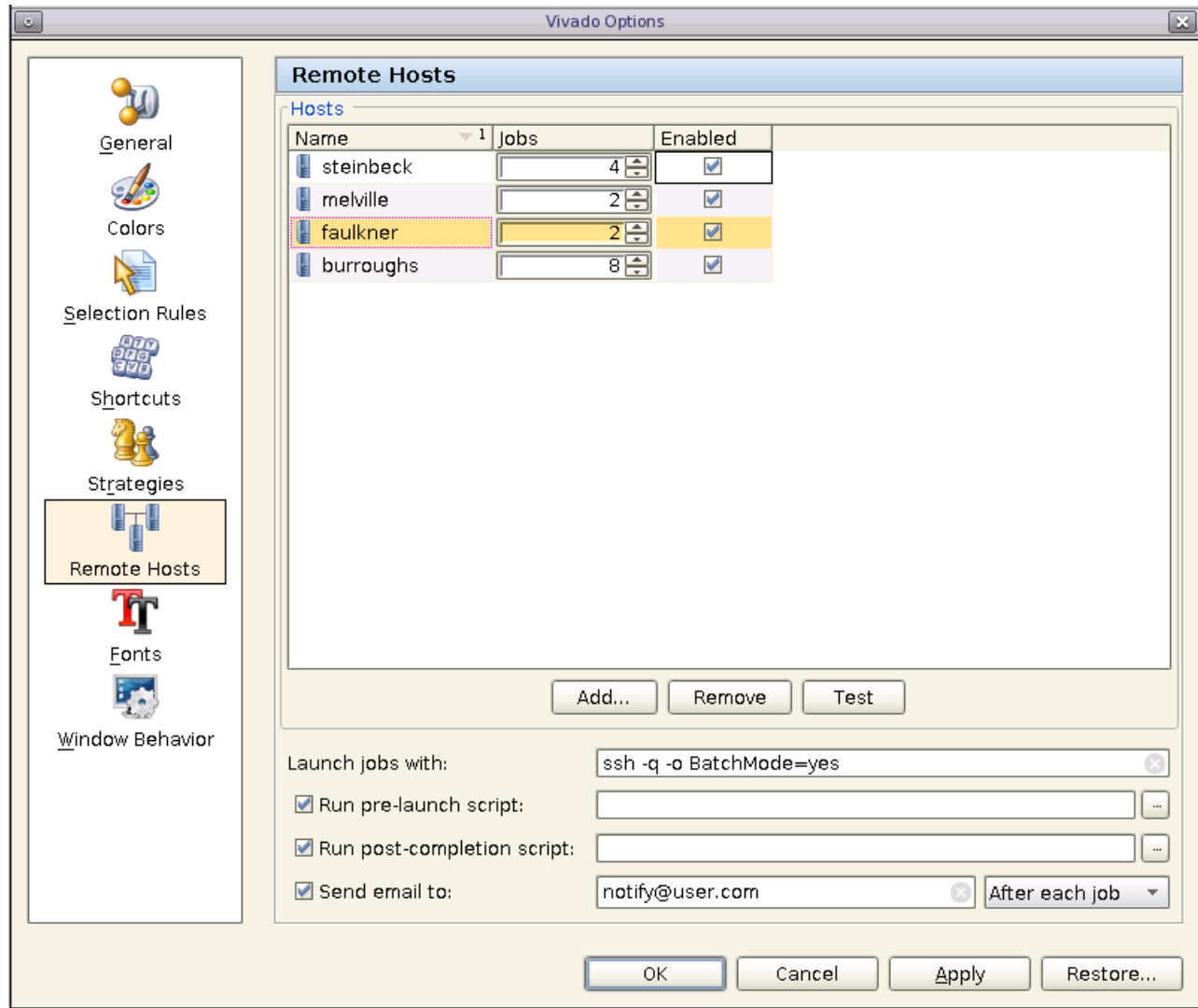


Figure A-2: Configuring Remote Hosts

To test the connection to the remote host:

1. Select one or more hosts.
2. Click **Test**.

The tool verifies that the server is available and that the configuration is properly set.



RECOMMENDED: Test each host to ensure proper set up before submitting runs to the host.

Removing Remote Hosts

To remove a remote host:

1. Select the remote host.
2. Click **Remove**.

Setting Up SSH Key Agent Forward

SSH configuration is accomplished with the following commands at a Linux terminal or shell:

Note: This is a one-time step. When successfully set-up, this step does not need to be repeated.

1. Run the following command at a Linux terminal or shell to generate a public key on your primary machine. Though not required, it is a good practice to enter (and remember) a private key phrase when prompted for maximum security.

```
ssh -keygen -t rsa
```

2. Append the contents of your public key to an `authorized_keys` file on the remote machine. Change `remote_server` to a valid host name:

```
cat ~/.ssh/id_rsa.pub | ssh remote_server "cat - >> ~/.ssh/authorized_keys"
```

3. Run the following command to prompt for your private key pass phrase, and enable key forwarding:

```
ssh -add
```

You should now be able to `ssh` to any machine without typing a password. The first time you access a new machine, it prompts you for a password. It does not prompt upon subsequent access.



TIP: *If you are always prompted for a password, contact your System Administrator.*

ISE Command Map

Tcl Commands and Options

Some command line options in the Xilinx® Vivado™ Integrated Design Environment (IDE) implementation are one-to-one equivalents of Xilinx Integrated Software Environment (ISE®) Design Suite commands.

Table B-2, [ISE Command Map](#), lists various ISE tool command line options, and their equivalent Vivado Design Suite Tcl command, and Tcl command options.

Note: For more information about Tcl commands, see the *Vivado Design Suite Tcl Command Reference Guide (UG835)* [Ref 12], or type `<command> -help`.

Table B-1: ISE Command Map

ISE Command	Vivado Tcl Command and Option
ngdbuild -p partname	link_design -part partname
ngdbuild -a (insert pads)	synth_design -no_iobuf (opposite)
ngdbuild -u (unexpanded blocks)	Enabled by default, generates critical warnings
ngdbuild -quiet	link_design -quiet
map -detail	opt_design -verbose
map -lc auto	Enabled by default in place_design
map -logic_opt	opt_design and phys_opt_design
map -mt	place_design automatically runs multi-threaded with four processors on Linux, or two processors on Windows
map -ntd	place_design -non_timing_driven
map -ol	place_design -effort_level
map -power	power_opt_design
map -u	link_design -mode out_of_context, opt_design -retarget (skip constant propagation and sweep)
par -pl	place_design -effort_level
par -rl	route_design -effort_level

Table B-1: ISE Command Map

ISE Command	Vivado Tcl Command and Option
par -mt	route_design automatically runs multi-threaded with four processors on Linux, or two processors on Windows
par -k	The re-entrant routing mode is the default mode of route_design
par -nopad	The -nopad behavior is the default behavior of Vivado. You must use report_io to obtain the PAD file report generated by PAR.
par -ntd	route_design -no_timing_driven

Table B-2: ISE Command Map

ISE Command	Vivado Tcl Command and Option
ngdbuild -p partname	link_design -part partname
ngdbuild -a (insert pads)	synth_design -no_iobuf (opposite)
ngdbuild -u (unexpanded blocks)	Enabled by default, generates critical warnings
ngdbuild -quiet	link_design -quiet
map -detail	opt_design -verbose
map -lc auto	Enabled by default in place_design
map -logic_opt	opt_design and phys_opt_design
map -mt	place_design automatically runs multi-threaded with four processors on Linux, or two processors on Windows
map -ntd	place_design -non_timing_driven
map -ol	place_design -effort_level
map -power	power_opt_design
map -u	link_design -mode out_of_context, opt_design -retarget (skip constant propagation and sweep)
par -pl	place_design -effort_level
par -rl	route_design -effort_level
par -mt	route_design automatically runs multi-threaded with four processors on Linux, or two processors on Windows
par -k	The re-entrant routing mode is the default mode of route_design
par -nopad	The -nopad behavior is the default behavior of Vivado. You must use report_io to obtain the PAD file report generated by PAR.
par -ntd	route_design -no_timing_driven

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.

References

Vivado Design Suite User Guides

1. *Vivado Design Suite User Guide: Design Flows Overview* ([UG892](#))
2. *Vivado Design Suite User Guide: Using the Vivado IDE* ([UG893](#))
3. *Vivado Design Suite User Guide: Using Tcl Scripting* ([UG894](#))
4. *Vivado Design Suite User Guide: System-Level Design Entry* ([UG895](#))
5. *Vivado Design Suite User Guide: Designing with IP* ([UG896](#))
6. *Vivado Design Suite User Guide: Synthesis* ([UG901](#))
7. *Vivado Design Suite User Guide: Using Constraints* ([UG903](#))

8. *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* ([UG906](#))
9. *Vivado Design Suite User Guide: Power Analysis and Optimization* ([UG907](#))
10. *Vivado Design Suite User Guide: Programming and Debugging* ([UG908](#))

Other Vivado Design Suite Documents

11. *7 Series FPGAs Clocking Resources* ([UG472](#))
12. *Vivado Design Suite Tcl Command Reference Guide* ([UG835](#))
13. *Vivado Design Suite Migration Methodology Guide* ([UG911](#))
14. *Vivado Design Suite Tutorial: Design Flows Overview* ([UG888](#))

Vivado Design Suite Video Tutorials

15. *Vivado Design Suite Video Tutorials* (<http://www.xilinx.com/training/vivado/index.htm>)

Vivado Design Suite Documentation

16. *Vivado Design Suite Documentation*
(www.xilinx.com/support/documentation/dt_vivado2013-1.htm)