See all versions of this document

Vivado Design Suite User Guide

Implementation

UG904 (v2020.2) February 26, 2021

XILINX

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
02/26/2021 Version 2020.2	
General Updates	General release updates.
08/25/2020 Version 2020.1	
Using Remote Hosts and Compute Clusters	Updated section.

Table of Contents

Revision History 2

Chapter 1: Preparing for Implementation

About the Vivado Implementation Process 5
Navigating Content by Design Process 8
Managing Implementation 8
Configuring, Implementing, and Verifying IP 14
Guiding Implementation with Design Constraints. 14
Using Checkpoints to Save and Restore Design Snapshots. 17

Chapter 2: Implementing the Design

Running Implementation in Non-Project Mode 18
Running Implementation in Project Mode. 22
Customizing Implementation Strategies 34
Launching Implementation Runs 40
Moving Processes to the Background. 42
Running Implementation in Steps 42
About Implementation Commands 44
Implementation Sub-Processes 44
Opening the Synthesized Design. 46
Logic Optimization 51
Power Optimization. 66
Placement. 68
Physical Optimization 85
Routing 100
Incremental Implementation 110

Chapter 3: Analyzing and Viewing Implementation Results

Monitoring the Implementation Run 130
Moving Forward After Implementation. 133
Viewing Messages 136
Viewing Implementation Reports. 138
Modifying Implementation Results 143

Vivado ECO Flow 169

Appendix A: Using Remote Hosts and Compute Clusters

Overview 189
Requirements 189
Manual Configuration 190
Cluster Configurations 192
Launching Jobs on Remote Hosts 197

Appendix B: ISE Command Map

Tcl Commands and Options 199

Appendix C: Implementation Categories, Strategy Descriptions, and Directive Mapping

Implementation Categories 200
Implementation Strategy Descriptions 200
Directives Used By opt_design and place_design in Implementation Strategies 202
Directives Used by phys_opt_design and route_design in Implementation Strategies 203

Appendix D: Additional Resources and Legal Notices

Xilinx Resources 206
Solution Centers 206
Documentation Navigator and Design Hubs 206
References 207
Training Resources 208
Please Read: Important Legal Notices 208

Chapter 1

Preparing for Implementation

About the Vivado Implementation Process

The Xilinx® Vivado® Design Suite enables implementation of the following Xilinx device architectures: Versal™ adaptive compute acceleration platform (ACAP), UltraScale™+, UltraScale™, and Xilinx 7 series FPGA. A variety of design sources are supported, including:

- RTL designs
- Netlist designs
- IP-centric design flows

Figure 1-1 shows the Vivado tools flow.

Vivado implementation includes all steps necessary to place and route the netlist onto device resources, within the logical, physical, and timing constraints of the design.

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].

SDC and XDC Constraint Support

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).

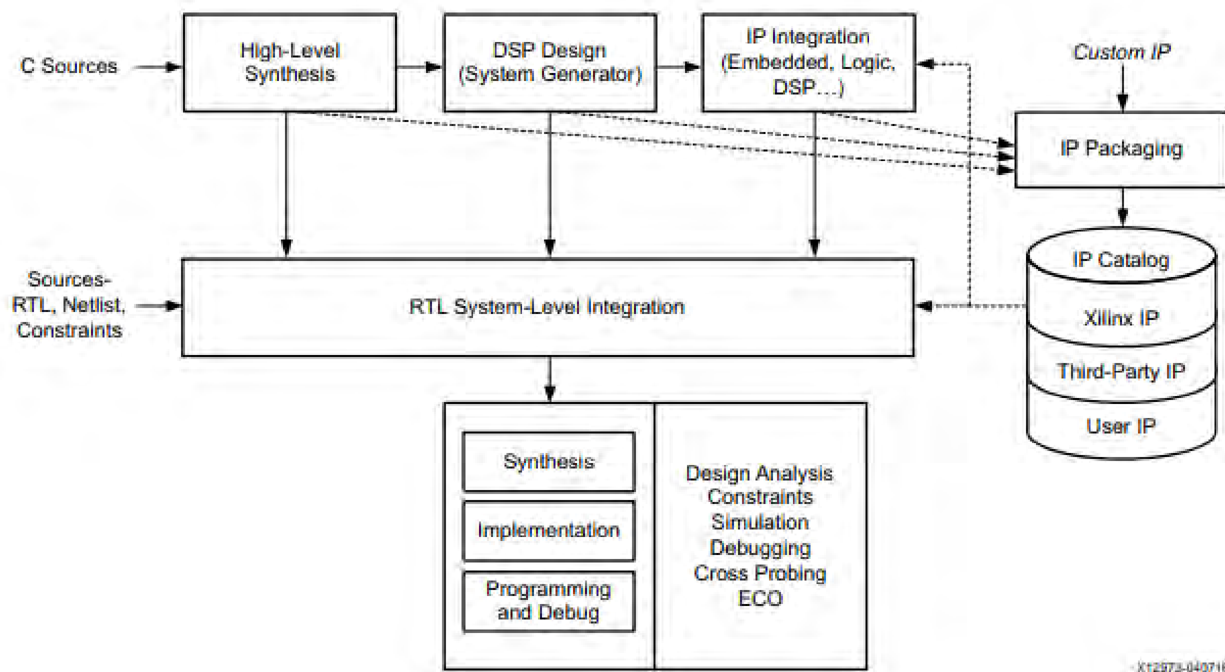


Figure 1-1: **Vivado Design Suite High-Level Design Flow**

Vivado Implementation Sub-Processes

The Vivado Design Suite implementation process transforms a logical netlist and constraints into a placed and routed design, ready for bitstream generation. The implementation process walks through the following sub-processes:

1. Opt Design:
   Optimizes the logical design to make it easier to fit onto the target Xilinx device.
2. Power Opt Design (optional):
   Optimizes design elements to reduce the power demands of the target Xilinx device.
3. Place Design:
   Places the design onto the target Xilinx device and performs fanout replication to improve timing.
4. Post-Place Power Opt Design (optional):
   Additional optimization to reduce power after placement.
5. Post-Place Phys Opt Design (optional):
   Optimizes logic and placement using estimated timing based on placement. Includes replication of high fanout drivers.
6. Route Design:
   Routes the design onto the target Xilinx device.

7. Post-Route Phys Opt Design (optional):
   Optimizes logic, placement, and routing using actual routed delays.
8. Write Bitstream:
   Generates a bitstream for Xilinx device configuration. Typically, bitstream generation follows implementation.

   For more information about writing the bitstream, see this link in the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 12].

Note: The Vivado Design Suite supports Module Analysis, which is the implementation of a part of a design to estimate performance. I/O buffer insertion is skipped for this flow to prevent over-utilization of I/O. For more information, search for "module analysis" in the *Vivado Design Suite User Guide: Hierarchical Design* (UG905) [Ref 2].

Multithreading with the Vivado Tools

On multiprocessor systems, Vivado tools use multithreading to speed up certain processes, including DRC reporting, static timing analysis, placement, and routing. The maximum number of simultaneous threads varies, depending on the number of processors and task. The maximum number of threads by task is:

- DRC reporting: 8
- Static timing analysis: 8
- Placement: 8
- Routing: 8
- Physical optimization: 8

The default number of maximum simultaneous threads is based on the OS. For Windows systems, the limit is 2; for Linux systems the default is 8. The limit can be changed using a parameter called `general.maxThreads`. To change the limit use the following Tcl command:

```
Vivado% set_param general.maxThreads <new limit>
```

where the new limit must be an integer from 1 to 8, inclusive.

Tcl example on a Windows system:

```
Vivado% set_param general.maxThreads 2
```

This means all tasks are limited to 2 threads regardless of number of processors or the task being executed. If the system has at least 8 processors, you can set the limit to 8 and allow each task to use the maximum number of threads.

```
Vivado% set_param general.maxThreads 8
```

To summarize, the number of simultaneous threads is the smallest of the following values:

- Maximum number of processors
- Limit of threads for the task
- General limit of threads

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 19] or type `<command> -help`.

Navigating Content by Design Process

Xilinx® documentation is organized around a set of standard design processes to help you find relevant content for your current development task. This document covers the following design processes:

Hardware, IP, and Platform Development

Creating the PL IP blocks for the hardware platform, creating PL kernels, subsystem functional simulation, and evaluating the Vivado® timing, resource use, and power closure. Also involves developing the hardware platform for system integration. Topics in this document that apply to this design process include:

- Vivado ECO Flow
- Configuring, Implementing, and Verifying IP
- Auto-Pipelining

Managing Implementation

The Vivado Design Suite includes a variety of design flows and supports an array of design sources. To generate a bitstream that can be downloaded onto a Xilinx 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. Implementation comprises:

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

Project Mode and Non-Project Modes

The Vivado Design Suite lets you run implementation with a project file (Project Mode) or without a project file (Non-Project Mode).

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.

Working in Project Mode

In Project Mode, a directory structure is created on disk to help you manage design sources, run results and reports, and track 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 tools automatically write checkpoint files into the local project directory at key points in the design flow.

To run implementation in Project Mode, you click the **Run Implementation** button in the IDE or use the `launch_runs` Tcl command. See this link in the *Vivado Design Suite User Guide: Design Flows Overview* (UG892) [Ref 1] for more information about using projects in the Vivado Design Suite.

Flow Navigator

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 appears in the left pane of the Vivado Design Suite main window. From the Flow Navigator you can assemble, implement, and validate the design and IP. It features a pushbutton interface to the entire implementation process to simplify the design flow. Figure 1-2 shows the implementation section of the Flow Navigator.

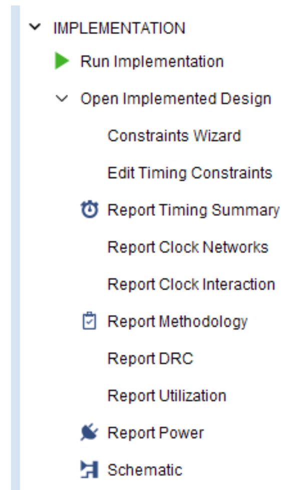


Figure 1-2: **Flow Navigator, Implementation Section**

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 design flow, see the Vivado Design Suite User Guide: Using the Vivado IDE (UG893) [Ref 3].*

Non-Project Mode

The Vivado tools also let you work with the design in memory, without the need for a project file and local directory. Working without a project file in the compilation style flow is called Non-Project Mode. 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, with the appropriate options for each implementation Tcl command.

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 *open* 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 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`
- `power_opt_design` (optional)
- `place_design`
- `phys_opt_design` (optional)
- `route_design`
- `phys_opt_design` (optional)
- `write_bitstream`

You can run implementation steps interactively in the Tcl Console, in 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 3]

Beginning the Implementation Flow

The implementation flow typically begins by loading a synthesized design into memory. Then the implementation flow can run, or the design can be analyzed and refined along with its constraints and the design can be reloaded after updates.

There are two ways to begin the implementation flow with a synthesized design:

- Run Vivado synthesis. In Project Mode, the synthesis run contains the synthesis results and those results are automatically used as the input for implementation run. In Non-Project Mode, the synthesis results are in memory after `synth_design` completes, and implementation can continue from that point.
- Load a synthesized netlist. Synthesized netlists can be used as the input design source, for example when using a third-party tool for synthesis.

To initiate implementation:

- In Project Mode, launch the implementation run.
- In Non-Project Mode run a script or interactive commands.

To analyze and refine constraints, the synthesized design is loaded without running implementation.

- In Project Mode, you accomplish this by opening the Synthesized Design, which is the result of the synthesis run.
- In Non-Project Mode, you use the `link_design` command to load the design.

You can also drive the implementation flow using design checkpoints (`.dcp`) in Non-Project Mode. Opening a checkpoint loads the design and restores it to its original state, which might include placement and routing data. This enables re-entrant implementation flows, such as loading a routed design and editing the routing, or loading a placed design and running multiple routes with different options.

Importing Previously Synthesized Netlists

The 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
- Structural SystemVerilog
- EDIF
- Xilinx NGC
- Synthesized Design Checkpoint (DCP)

IMPORTANT: *NGC format files are not supported in the Vivado Design Suite for UltraScale and later devices. It is recommended that you regenerate the IP using the Vivado Design Suite IP customization tools with native output products. Alternatively, convert_ngc Tcl utility to convert NGC files to EDIF or Verilog formats. However, Xilinx recommends using native Vivado IP rather than XST-generated NGC format files going forward.*

IMPORTANT: *When using IP in Project Mode or Non-Project Mode, always use the XCI file and not the DCP file. This ensures that IP output products are used consistently during all stages of the design flow. If the IP was synthesized out-of-context and already has an associated DCP file, the DCP file is automatically used and the IP is not re-synthesized. For more information, this link in the Vivado Design Suite User Guide: Designing with IP (UG896) [Ref 4].*

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 6].

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 Vivado synthesis before implementation can begin. The Vivado IDE manages this automatically if you attempt to run implementation on an un-synthesized design. The tools allow you to run synthesis first.*

For information on running Vivado synthesis, see the *Vivado Design Suite User Guide: Synthesis* (UG901) [Ref 8].

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. You can open a synthesized design checkpoint file using the `open_checkpoint` command.

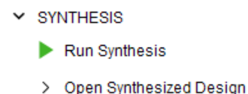
For more information, see Opening the Synthesized Design in Chapter 2.

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 Run** from the context menu.

Configuring, Implementing, and Verifying IP

For information on importing IP into your design prior to synthesis, see this link in the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 4].

Guiding Implementation with Design Constraints

There are three types of design constraints, physical constraints, timing constraints and power constraints. These are defined below.

Physical Constraints Definition

Physical constraints define a relationship between logical design objects and device resources such as:

- Package pin placement.
- Absolute or relative placement of cells, including Block RAM, DSP, LUT, and flip-flops.
- Floorplanning constraints that assign cells to general regions of a device.
- Device configuration settings.

Timing Constraints Definition

Timing constraints define the frequency requirements for the design, and are written in Xilinx Design Constraints (XDC) which is based on 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.

Power Constraints Definition

Power constraints define the settings needed for accurate power analysis. These settings include:

- Operating conditions such as voltage settings, power and current budgets, and operating environment details.
- Switching activity rates for:
  - Design objects: individual nets and pins.
  - Design object types such as block RAMs, DSPs, and transceivers.
  - Global set and reset signals.

Vivado power analysis uses timing constraints to determine switching rates and applies vectorless propagation to determine toggle rates throughout the design. Without power constraints, a default 12.5% toggle rate is used. However, applying accurate switching activity to override defaults is essential for accurate power calculations.

For further information see *Vivado Power Analysis and Optimization* (UG907) [Ref 11].

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 this link in the *ISE to Vivado Design Suite Migration Guide* (UG911) [Ref 20].

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 in Project Mode. The set contains design constraints captured in XDC or Tcl 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 devices for the same project. Different physical and timing constraints might be needed for different target devices.
- 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 and report_methodology on the synthesized design. Fix problematic constraints before implementation!*

For more information on defining and working with constraints that affect placement and routing, see this link in the *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 9].

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 HDL attributes, and are not available in XDC. In those cases, the constraint must be specified as an attribute in the HDL source file. For example, Relatively Placed Macros (RPMs) must be defined using HDL attributes. An RPM is a set of logic elements (such as FF, LUT, DSP, and RAM) with relative placements.

You can define RPMs using `U_SET` and `HU_SET` attributes and define relative placements using Relative Location Attributes.

For more information about Relative Location Constraints, see this link in the *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 9].

For more information on constraints that are *not* supported in XDC, see the *ISE to Vivado Design Suite Migration Guide* (UG911) [Ref 20].

Using Checkpoints to Save and Restore Design Snapshots

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. A checkpoint is a snapshot of a design at a specific point in the flow.

This design checkpoint file includes:

- 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.

IMPORTANT: *In Project Mode, the Vivado design tools automatically save and restore checkpoints as the design progresses. In Non-Project Mode, you must save checkpoints at appropriate stages of the design flow, otherwise, progress is lost.*

Writing Checkpoint Files

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

The related Tcl command is `write_checkpoint`.

Reading Checkpoint Files

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

The design checkpoint is opened as a separate in-memory design.

The related Tcl command is `open_checkpoint`.

Implementing the Design

Running Implementation in Non-Project Mode

To implement the synthesized design or netlist onto the targeted Xilinx® devices in Non-Project Mode, you must run the Tcl commands corresponding to the Implementation sub-processes:

- Opt Design (`opt_design`): Optimizes the logical design to make it easier to fit onto the target Xilinx device.
- Power Opt Design (`power_opt_design`) (optional): Optimizes design elements to reduce the power demands of the target Xilinx device.
- Place Design (`place_design`): Places the design onto the target Xilinx device and replicates logic to improve timing.
- Post-Place Power Opt Design (`power_opt_design`) (optional): Additional optimization to reduce power after placement.
- Post-Place Phys Opt Design (`phys_opt_design`) (optional): Optimizes logic and placement using estimated timing based on placement. Includes replication of high fanout drivers.
- Route Design (`route_design`): Routes the design onto the target Xilinx device.
- Post-Route Phys Opt Design (`phys_opt_design`) (optional): Optimizes logic, placement, and routing using actual routed delays.
- Write Bitstream (`write_bitstream`): Generates a bitstream for Xilinx device configuration except for Versal ACAP devices. Typically, bitstream generation follows implementation.
- Write Device Image (write_device_image): Generates a a programmable device image for programming a Versal device.

For more information about writing the bitstream or creating a device image, see this link in the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 12].

These steps are collectively known as *implementation*.

Enter the commands in any of the following ways:

- In the Tcl Console from the Vivado® IDE.
- From the Tcl prompt in the Vivado Design Suite Tcl shell.
- Using a Tcl script with the implementation commands and source the script in the Vivado Design Suite.

Non-Project Mode Example Script

The script below is an example of running implementation in Non-Project Mode. Assuming the script is named `run.tcl`, you would call the script using the source command in the Tcl shell.

Note: The `read_xdc` step reads XDC constraints from the XDC files and applies constraints to design objects. Therefore all netlist files must be read into Vivado and `link_design` should be run before `read_xdc` to ensure that the XDC constraints can be applied to their intended design objects.

```
source run.tcl
```

```
# 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

# 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 using the default directive and save a checkpoint
# It is recommended to save progress at certain intermediate steps
# The placed checkpoint can also be routed in multiple runs using different options
place_design -directive Default
write_checkpoint post_place.dcp

# Step 6: Route the design with the AdvancedSkewModeling directive. For more
information
# on router directives type 'route_design -help' in the Vivado Tcl Console
route_design -directive AdvancedSkewModeling

# 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, page 19 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 Design
- Step 6: Route the Design
- Step 7: Run Required Reports
- Step 8: Save the Design Checkpoint

Step 1: Read Design Source Files

EDIF netlist design sources are read into memory through use of the `read_edif` command. Non-Project Mode also supports an RTL design flow, which allows you to read source files and run synthesis before implementation.

Use the `read_checkpoint` command to add synthesized design checkpoint files as sources.

The `read_*` Tcl commands are designed for use with Non-Project Mode. The `read_*` Tcl commands allow the Vivado tools to read a file on the disk and build the in-memory design without copying the file or creating a dependency on the file.

This approach makes Non-Project Mode highly 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

The Vivado tools build an in-memory view of the design using `link_design`. The `link_design` command combines the netlist based source files read into the tools with the Xilinx part information, to create a design database in memory.

There are two important `link_design` options:

- The `-part` option specifies the target device.

- The `-top` option specifies the top design for implementation. If the top-level netlist is EDIF and the `-top` option is not specified, the Vivado tools will use the top design embedded in the EDIF netlist. If the top-level netlist is not EDIF but structural Verilog, the `-top` option is required. The `-top` option can also be used to specify a submodule as the top, for example when running the Module Analysis flow to estimate performance and utilization.

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 tools, 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 14.

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

TIP: *Although 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

Logic optimization is run 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 51.

Step 5: Place the Design

The `place_design` command places the design. For more information, see Placement, page 68. After placement, the progress is saved to a design checkpoint file using the `write_checkpoint` command.

Step 6: Route the Design

The `route_design` command routes the design. For more information, see Routing, page 100.

Step 7: Run Required Reports

The `report_timing_summary` command runs timing analysis and generates a timing report with details of timing violations. The `report_utilization` command generates a summary of the percentage of device resources used along with other utilization statistics. In Non-Project Mode, you must use the appropriate Tcl command to specify each report that you want to create. Each reporting command supports the `-file` option to direct output to a file.

See this link the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 19] for further information on the `report_timing_summary` command and this link for further information on `report_utilization` command.

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 138.

Step 8: Save the Design Checkpoint

Saves the in-memory design into a design checkpoint file. The saved in-memory design includes the following:

- Logical 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 Using Checkpoints to Save and Restore Design Snapshots.

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.*

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 and Compute Clusters.

Defining Implementation Runs

To define an implementation run:

1. From the main menu, select **Flow > Create Runs**.

   Alternatively, in the Flow Navigator, select **Create Implementation Runs** from the Implementation popup menu. Or, in the Design Runs window, select **Create Runs** from the popup menu.

   The Create New Runs wizard opens.

2. Select **Implementation** on the first page of the Create New Runs wizard, and click **Next**.
3. The Configure Implementation Runs screen appears, as shown in Figure 2-1. Specify settings as described in the steps below the figure.

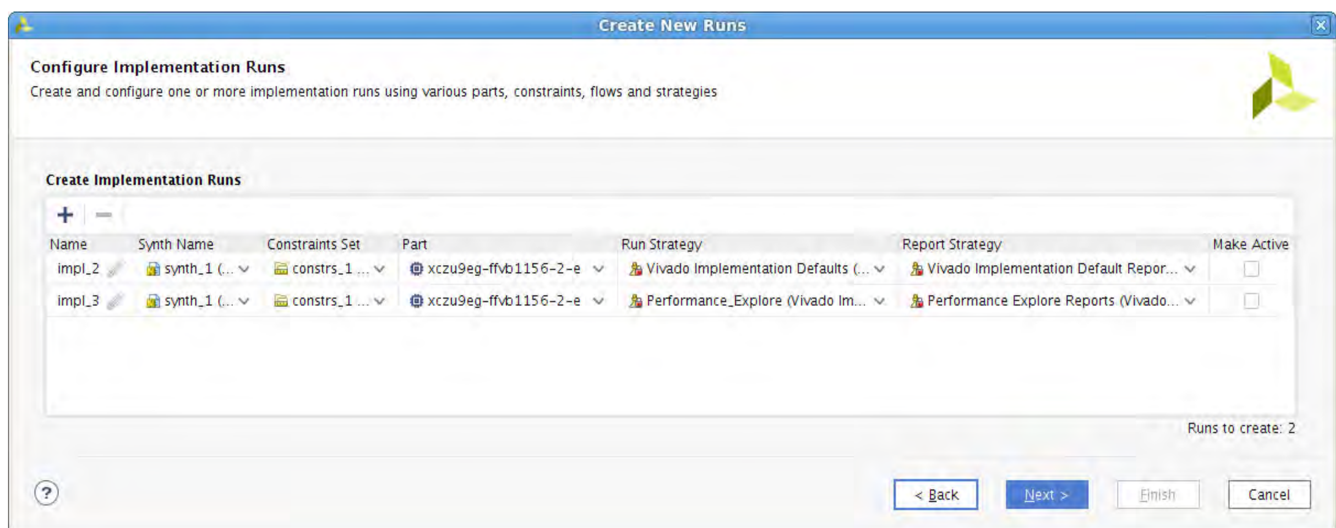


Figure 2-1: **Configure Implementation Runs**

a. In the **Name** column, enter a name for the run in the Configure Implementation Runs dialog box or accept the default name.

b. Select a **Synth Name** to choose the synthesis run that will generate (or that has already generated) the synthesized netlist to be implemented. The default is the currently active synthesis run in the Design Runs window. For more information, see Appendix C, Implementation Categories, Strategy Descriptions, and Directive Mapping.

   Note: In the case of a netlist-driven project, the Create Run command does not require the name of the synthesis run.

   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 8].

c. Select a **Constraints 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 9].

d. 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 this link in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895) [Ref 6].

TIP: *To create runs with different constraint sets or target parts, use the* ***Create New Runs*** *command. To change these values on existing runs, select the run in the Design Runs window and edit the Run Properties.*

   For more information, see Changing Implementation Run Settings, page 29.

e. Select a **Strategy**.

   Strategies are a defined set of Vivado implementation feature options that control the implementation results. Vivado Design Suite includes a set of pre-defined strategies. You can also create your own implementation strategies.

   Select from among the strategies shown in Appendix C, Implementation Categories, Strategy Descriptions, and Directive Mapping. The strategies are broken into categories according to their purposes, with the category name as a prefix. The categories are shown in Appendix C.

   For more information see Defining Implementation Strategies, page 35.

TIP: *The optimal strategy can change between designs and software releases.*

The purpose of using Performance strategies is to improve design performance at the expense of run time. You should always try to meet timing goals, using the Vivado implementation defaults first, before choosing a Performance strategy. This ensures that your design has sufficient margin for absorbing timing closure impact due to design changes. But if your design goals cannot be met, and if increased run time is acceptable, the `Performance_Explore` strategy is a good first choice. It covers all types design types.

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

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 29*.

f. Click **More** to define additional runs. By default, the next strategy in the sequence is automatically chosen. Specify names and strategies for the added runs. See Figure 2-1, above.

g. Use the **Make Active** check box to select the runs you wish to initiate.

h. Click **Next**.

4. The Launch Options screen appears, as shown in Figure 2-2. Specify options as described in the steps below the figure.

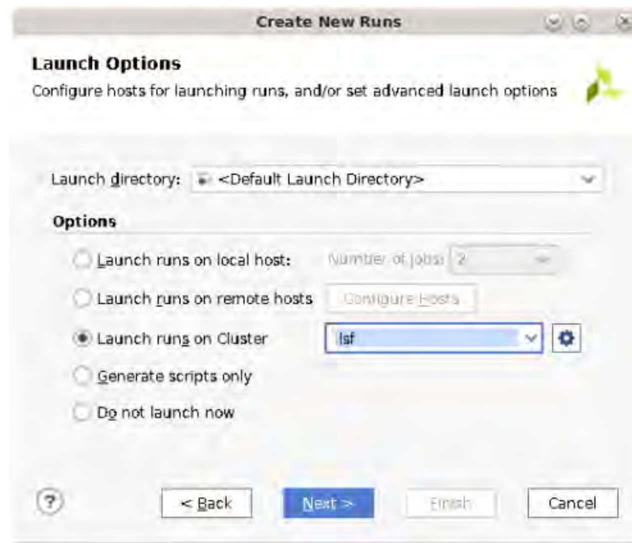


Figure 2-2: **Implementation Launch Options**

Note: The **Launch runs on remote hosts** and **Launch runs on Cluster** options shown in Figure 2-2 are Linux-only. They are not visible on Windows machines.

a. Specify the Launch directory, the location at which implementation run data is created and stored.

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.*

b. Use the radio buttons and drop-down options to specify settings appropriate to your project. Choose from the following:

   - Select the **Launch runs on local host** option if you want to launch the run on the local machine.
   - Use the **Number of jobs** drop-down menu to define the number of local processors to use when launching multiple runs simultaneously.
   - Select **Launch runs on remote hosts** (Linux only) if you want to use remote hosts to launch one or more jobs.
   - Use the **Configure Hosts** button to configure remote hosts. For more information, see Appendix A, Using Remote Hosts and Compute Clusters.
   - Select **Launch runs on Cluster** (Linux only) if you want to use a compute cluster command to launch one or more jobs. Use the drop down menu to select one of the natively supported Vivado Clusters (lsf, sge or slurm) or a User Defined Cluster that has been added previously.
   - Select the **Generate scripts only** option if you want to export and create the run directory and run script but do not want the run script to launch at this time. The script can be run later outside the Vivado IDE tools.
   - Select **Do not launch now** if you want to save the new runs, but you do not want to launch or create run scripts at this time.

5. Click **Next** to review the Create New Runs Summary.
6. Click **Finish** to create the defined runs and execute the specified launch options.

New runs are added to the Design Runs window. See Using 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 (see Figure 2-3) if it is not already open.

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 this link in the *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893) [Ref 3].

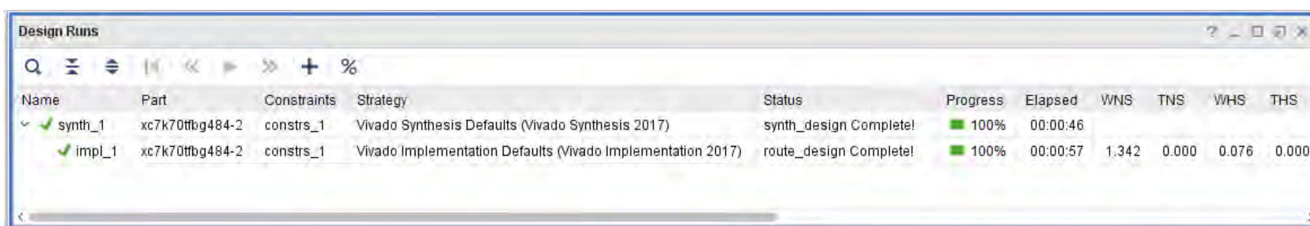


Figure 2-3: **Design Runs Window**

Run Status

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

- The run has not been started.
- The run is in progress.
- The run is complete.
- The run is out-of-date.

The Design Runs window reports start and elapsed run times.

Run Times

The Design Runs window reports start time and elapsed time for the runs.

Run Timing Results

The Design Runs window reports timing results for implementation runs including WNS, TNS, WHS, THS, and TPWS.

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, shown in Figure 2-4, below.

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 this link in the *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893) [Ref 3].

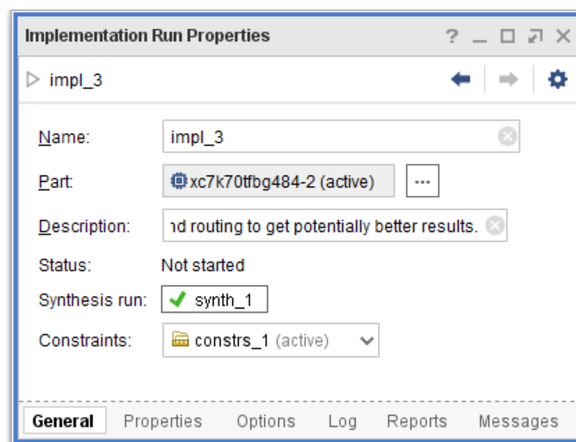


Figure 2-4: **Implementation Run Properties Window**

Specifying Design Run Settings

Specify design run settings in the Design Run Settings dialog box, shown in Figure 2-5. To open the Design Run Settings dialog box:

1. Right-click a run in the Design Runs window.
2. Select **Change Run Settings** from the popup menu to open the Design Run Settings dialog box, shown in Figure 2-5.

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 33*.

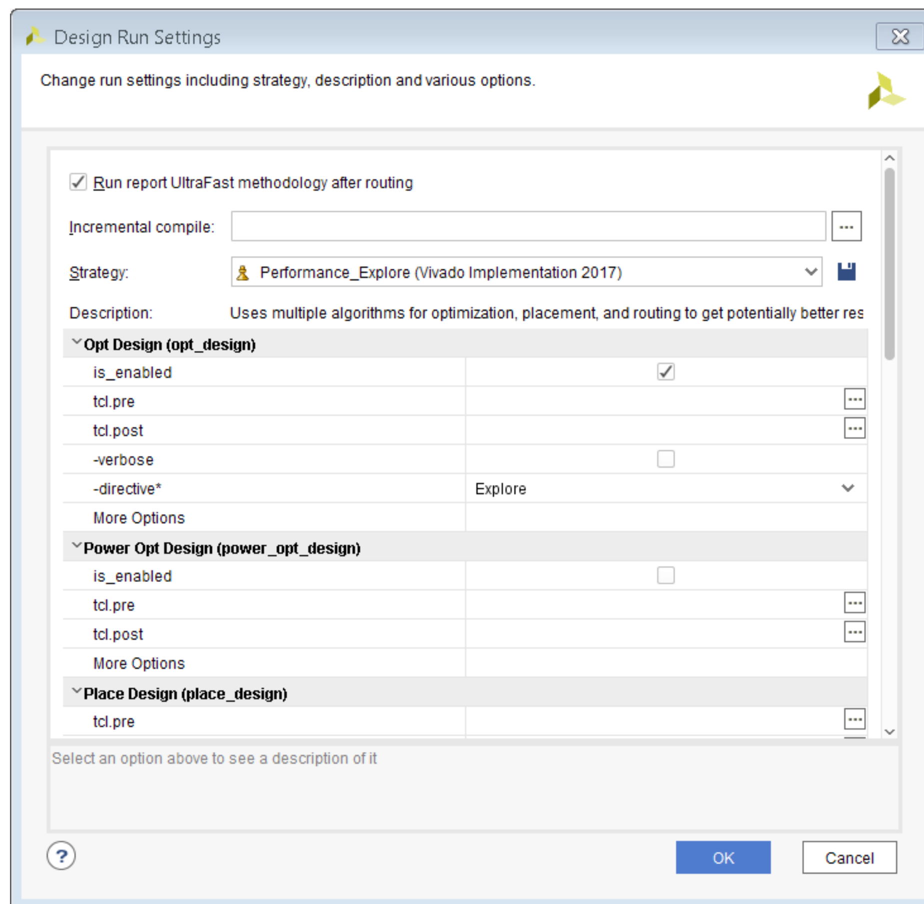


Figure 2-5: **Design Run Settings**

The Design Run Settings dialog box displays the following:

- The implementation strategy currently employed by the run.
- The command options associated with that strategy for each step of the implementation process. The three command options are described below.

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 Implementation Strategies, page 35.

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 (`opt_design`)
- Power Opt Design (`power_opt_design`) (optional)
- Place Design (`place_design`)
- Post-Place Power Opt Design (`power_opt_design`) (optional)
- Post-Place Phys Opt Design (`phys_opt_design`) (optional)
- Route Design (`route_design`)
- Post-Route Phys Opt Design (`phys_opt_design`) (optional)
- Write Bitstream (`write_bitstream`)

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

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 deselect a check box to enable or disable options.

  Note: The most common options for each implementation command are available through the check boxes. Add other supported command options using the More Options field. Syntax: precede option names with a hyphen and separate options from each other with a space.

- 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 this link in the *Vivado Design Suite User Guide: Using Tcl Scripting* [Ref 5].

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 (shown in Figure 2-3).

- If the status of the run is **Not Started**, the run begins immediately.
- If the status of the run is **Error**, the tools reset 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 tools prompt 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.

As shown in Figure 2-6, the Vivado tools prompt you to confirm the **Reset Runs** command, and optionally delete the generated files from the `run` directory.

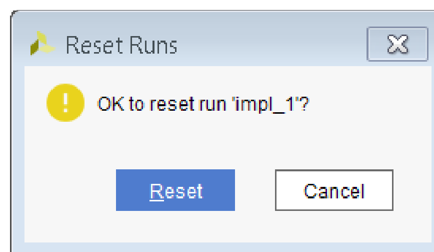


Figure 2-6: **Reset Run Prompt**

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.

As shown in Figure 2-7, the Vivado tools prompt 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.*

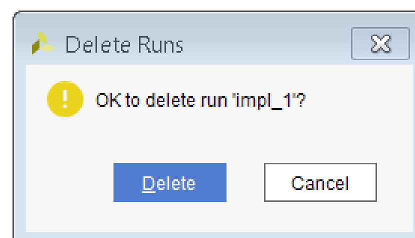


Figure 2-7: **Delete Runs Prompt**

Customizing Implementation Strategies

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

Figure 2-8 shows the Implementation Settings view in the Settings dialog box. To open this dialog box from the Vivado IDE, select **Tools > Settings** from the main menu.

TIP: *The 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

You can also access Implementation Settings for the active implementation run by selecting **Settings** at the top of the Flow Navigator, and then clicking the Implementation category.

The Settings dialog box, shown in Figure 2-8, contains the following fields:

- Default Constraint Set:
  Select the constraint set to be used by default for the implementation run.
- Report Settings:
  Use this menu to select the report strategy. You can choose from a preset report strategy or define your own strategy to choose which reports to run at each design step.
- Incremental Compile:
  Specify the Incremental Compile checkpoint, if desired.
- Strategy:
  Select the strategy to use for the implementation run. The Vivado Design Suite includes a set of pre-defined strategies. You can also create your own implementation strategies and save changes as new strategies for future use. For more information see Defining Implementation Strategies.
- Description:
  Describes the selected implementation strategy. The description of user-defined strategies can be changed by entering a new descriptions. The description of Vivado tools standard implementation strategies cannot be changed.

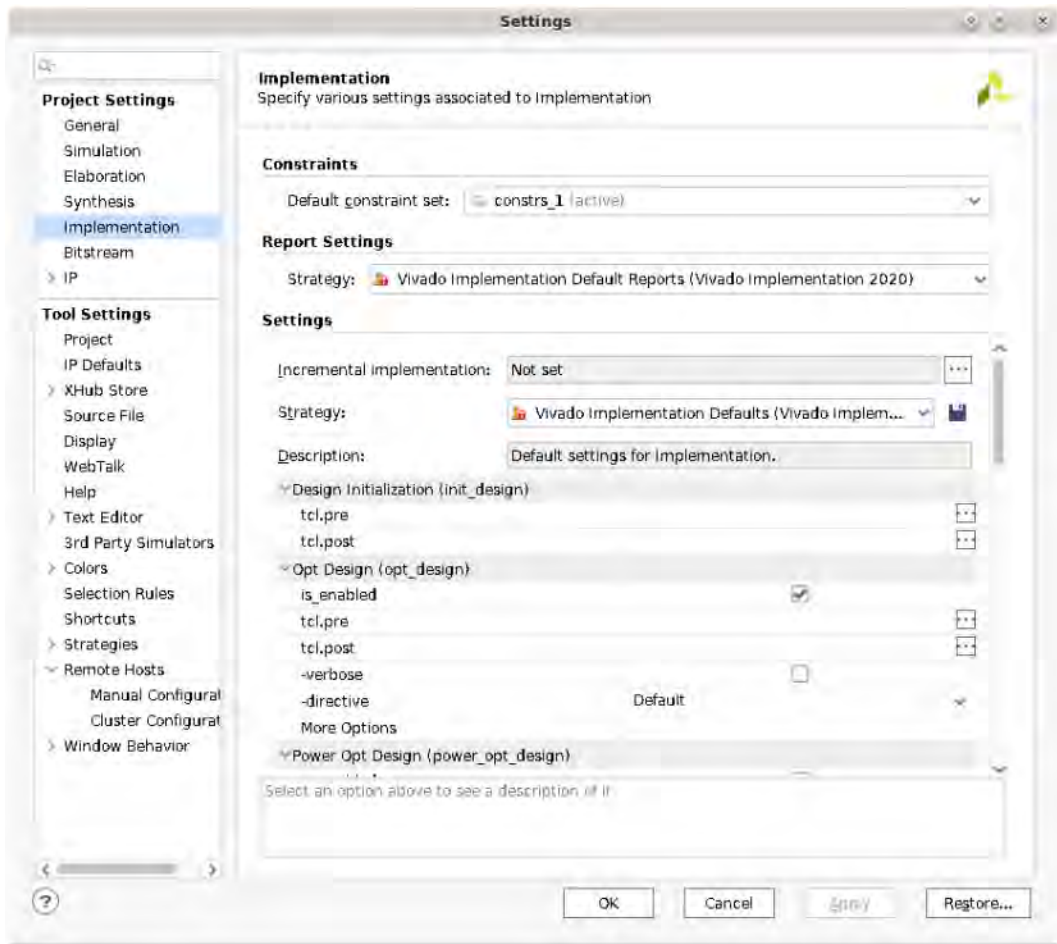


Figure 2-8: **Implementation Settings**

Defining Implementation Strategies

A run 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.

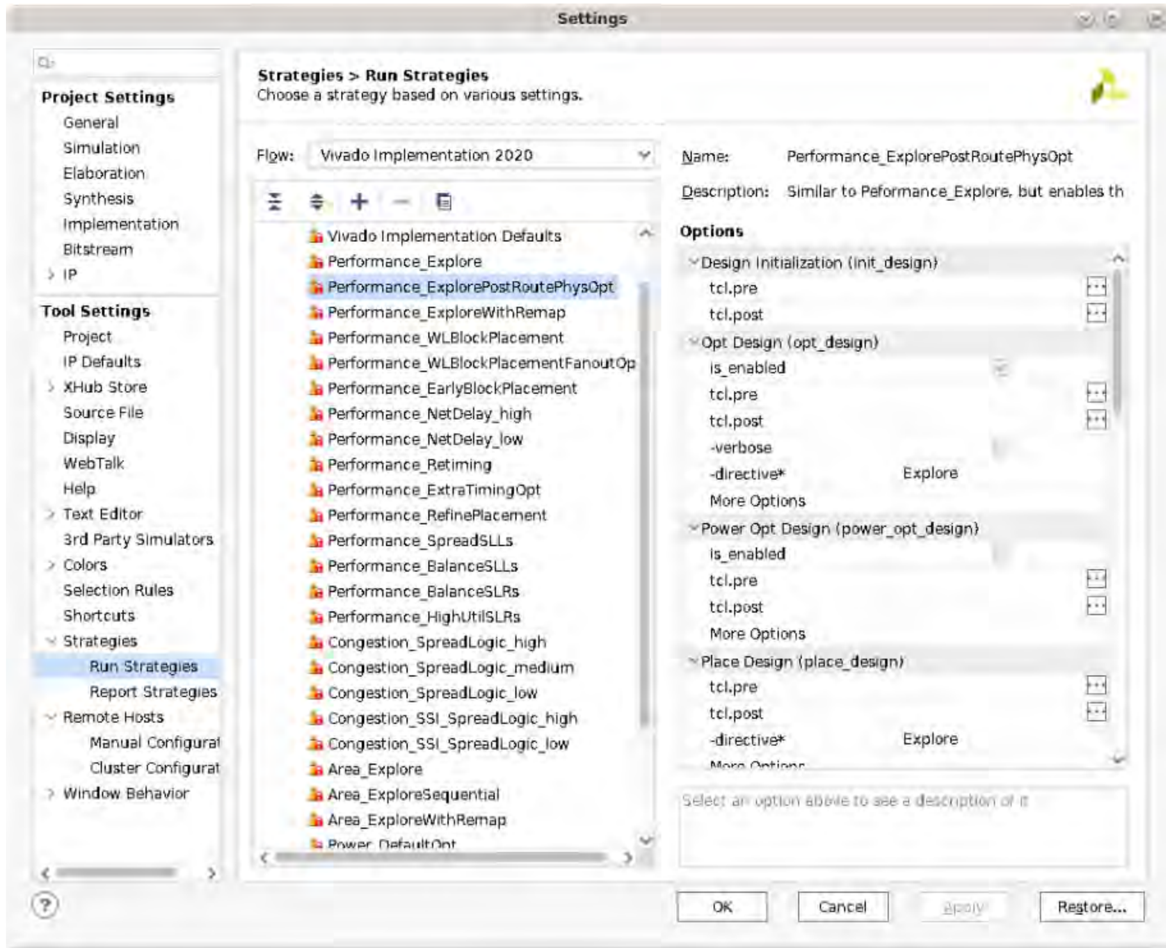


Figure 2-9: **Default Implementation Strategies**

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

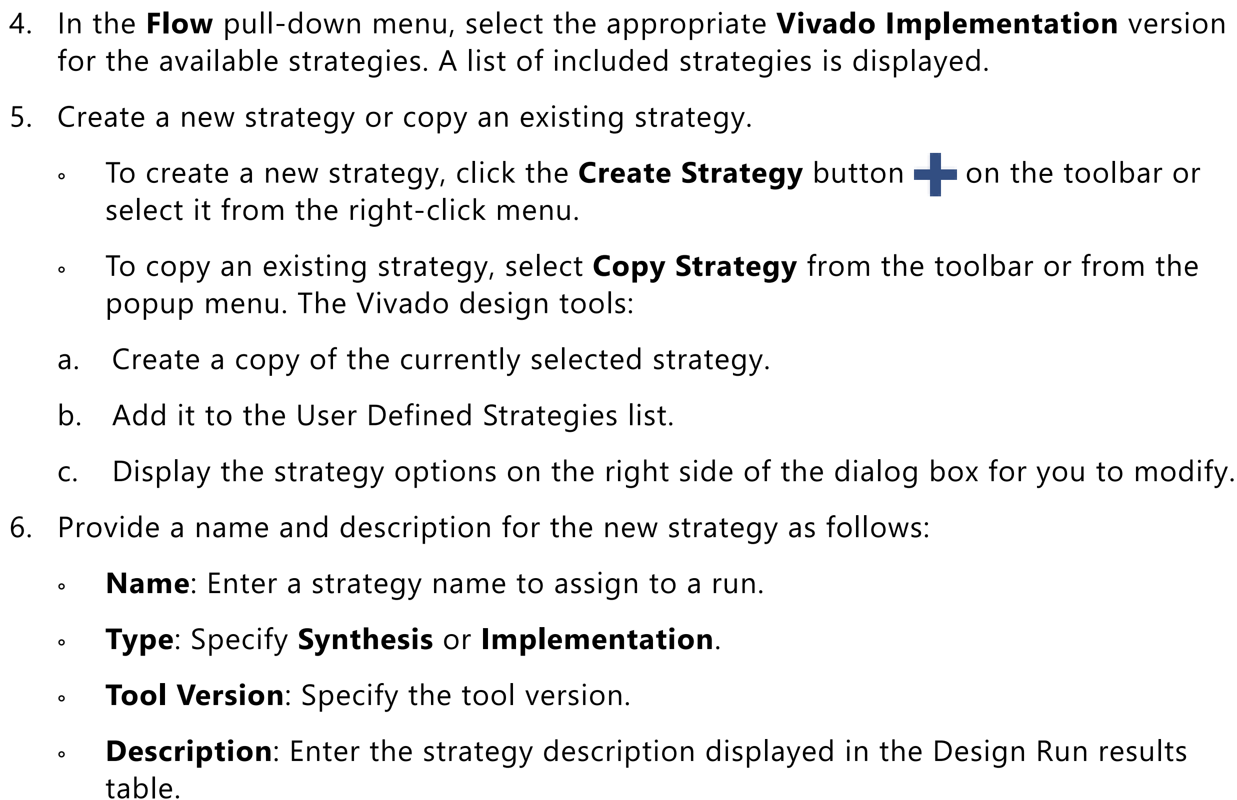
To access the currently defined run strategies, select **Tools > Settings** in the Vivado IDE main menu.

Reviewing, Copying, and Modifying Strategies

To review, copy, and modify run strategies:

1. Select **Tools > Settings** from the main menu.

2. Select **Strategies** in the left-side panel.
3. Select **Run Strategies** to review, copy, or modify run strategies. The Run Strategies page (shown in Figure 2-9) contains a list of pre-defined run strategies for various tools and release versions.

   Note: For information on reviewing, copying, or modifying Report Strategies, see this link in *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 10].
4. In the **Flow** pull-down menu, select the appropriate **Vivado Implementation** version for the available strategies. A list of included strategies is displayed.
5. Create a new strategy or copy an existing strategy.
   - To create a new strategy, click the **Create Strategy** button  on the toolbar or select it from the right-click menu.
   - To copy an existing strategy, select **Copy Strategy** from the toolbar or from the popup menu. The Vivado design tools:

   a. Create a copy of the currently selected strategy.

   b. Add it to the User Defined Strategies list.

   c. Display 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 **Implementation**.
   - **Tool Version**: Specify the tool version.
   - **Description**: Enter the strategy description displayed in the Design Run results table.

7. Edit the **Options** for the various implementation steps:
   - Design Initialization (`init_design`)
   - Opt Design (`opt_design`)
   - Power Opt Design (`power_opt_design`) (optional)
   - Place Design (`place_design`)
   - Post-Place Power Opt Design (`power_opt_design`) (optional)
   - Post-Place Phys Opt Design (`phys_opt_design`) (optional)
   - Route Design (`route_design`)
   - Post-Route Phys Opt Design (`phys_opt_design`) (optional)
   - Write Bitstream (`write_bitstream`) (all devices except Versal)
   - Write Device Image (`write_device_image`) (Versal devices)

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

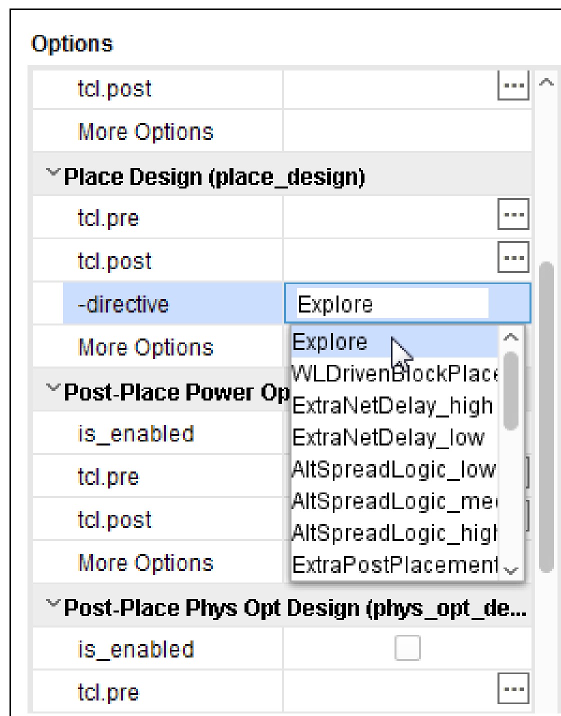


Figure 2-10: **Edit Implementation Steps**

8. Click the right-side column of a specific option to modify command options. See Figure 2-10 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 this link in the *Vivado Design Suite User Guide: Using Tcl Scripting* (UG894) [Ref 5].

     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]
```

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

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

- Linux OS

  `$HOME/.Xilinx/Vivado/strategies`

- Windows

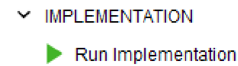
  `C:\Users\<username>\AppData\Roaming\Xilinx\Vivado\strategies`

Sharing Run 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, and `<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, shown in Figure 2-11.

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

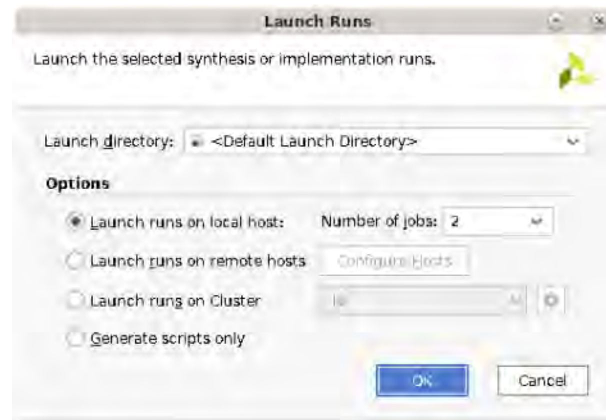


Figure 2-11: **Launch Selected Implementation Runs**

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.
   - Select the **Launch runs on local host** option if you want to launch the run on the local machine.
   - Use the **Number of jobs** drop-down menu to define the number of local processors to use when launching multiple runs simultaneously.
   - Select **Launch runs on remote hosts** (Linux only) if you want to use remote hosts to launch one or more jobs.
   - Use the **Configure Hosts** button to configure remote hosts. For more information, see Appendix A, Using Remote Hosts and Compute Clusters.
   - Select **Launch runs using LSF** (Linux only) if you want to use LSF (Load Sharing Facility) `bsub` command to launch one or more jobs. Use the **Configure LSF** button to set up the `bsub` command options and test your LSF connection.

TIP: *LSF, the Load Sharing Facility, is a subsystem for submitting, scheduling, executing, monitoring, and controlling a workload of batch jobs across compute servers in a cluster.*

   - Select the **Generate scripts only** option if you want to export and create the run directory and run script but do not want the run script to launch at this time. The script can be run later outside the Vivado IDE tools.

Moving Processes to 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, shown in Figure 2-12, lets you move this preparation to the background.

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.*

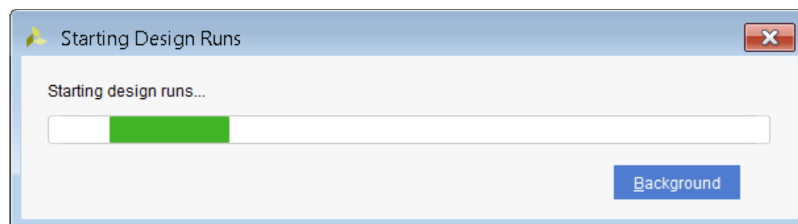


Figure 2-12: **Starting Run - Background Process**

Running Implementation in Steps

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

- Opt Design (`opt_design`)
- Power Opt Design (`power_opt_design`) (optional)
- Place Design (`place_design`)
- Post-Place Power Opt Design (`power_opt_design`) (optional)
- Post-Place Phys Opt Design (`phys_opt_design`) (optional)
- Route Design (`route_design`)
- Post-Route Phys Opt Design (`phys_opt_design`) (optional)
- Write Bitstream (`write_bitstream`) (all devices except Versal)
- Write Device Image (`write_device_image`) (Versal devices)

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. Right-click a run in the **Design Runs** window and select **Launch Next Step: <*Step*>** or **Launch Step To** from the popup menu shown in Figure 2-13.

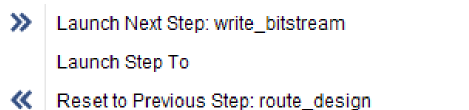


Figure 2-13: **Popup Menu in Design Runs Window**

   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 device.
   - Power Opt Design:
     Optimizes elements of the design to reduce power demands of the implemented device.
   - Place Design:
     Places the design onto the target Xilinx device.
   - Post-Place Power Opt Design:
     Additional optimization to reduce power after placement.
   - Post-Place 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.
   - Post-Route Phys Opt Design:
     Optimizes logic, placement, and routing, using actual routed delays.
   - Write Bitstream (all devices except Versal):
     Generates a bitstream for Xilinx device configuration. Although not technically part of an implementation run, bitstream generation is available as an incremental step.
   - Write Device Image (Versal devices):
     Generates a programmable device image for programming a Versal device.

2. Repeat **Launch Next Step: <*Step*>** or **Launch Step To** as needed to move the design through implementation.

3. 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.

About Implementation Commands

The Xilinx® Vivado® Design Suite 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 22.

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 19], or type `<command> -help`.

For more information, see Running Implementation in Non-Project Mode, page 18.

Implementation Sub-Processes

In Project Mode, the implementation commands are run in a fixed order. In Non-Project Mode the commands can be run in a similar order, but can also be run repeatedly, iteratively, and in a different sequence than in Project Mode.

IMPORTANT: *Implementation Commands are re-entrant.*

Implementation commands are re-entrant, which means that when an implementation command is called in Non-Project Mode, it reads the design in memory, performs its tasks, and writes the resulting design back into memory. This provides more flexibility when running in Non-Project Mode.

Examples:

- `opt_design` followed by `opt_design -remap`
  The Remap operation occurs on the `opt_design` results.
- `place_design` called on a design that contains some placed cells
  The existing cell placement is used as a starting point for `place_design`.
- `route_design` called on a design that contains some routing
  The existing routing is used as a starting point for `route_design`.
- `route_design` called on a design with unplaced cells
  Routing fails because cells must be placed first.
- `opt_design` called on a fully-placed and routed design
  Logic optimization might optimize the logical netlist, creating new cells that are unplaced, and new nets that are unrouted. Placement and routing might need to be rerun to finish implementation.

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 to make it easier to fit onto the target Xilinx device.
- Power Opt Design (optional):
  Optimizes design elements to reduce the power demands of the target Xilinx device.
- Place Design:
  Places the design onto the target Xilinx device.
- Post-Place Power Opt Design (optional):
  Additional optimization to reduce power after placement.
- Post-Place Phys Opt Design (optional):
  Optimizes logic and placement using estimated timing based on placement. Includes replication of high fanout drivers.
- Route Design:
  Routes the design onto the target Xilinx device.
- Post-Route Phys Opt Design:
  Optimizes logic, placement, and routing using actual routed delays (optional).
- Write Bitstream:
  Generates a bitstream for Xilinx device configuration (except Versal device).
- Write Device Image:
  Generates a programmable device image for programming a Versal device.

Note: Although not technically part of an implementation run, Write Bitstream and Write Device Image are 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. The following table provides a list of sub-processes and their associated Tcl commands.

Table 2-1: **Implementation Sub-processes and Associated Tcl Commands**

Sub-Process	Tcl Command
Open Synthesized Design	synth_design
	open_checkpoint
	open_run
	link_design
Opt Design	opt_design
Power Opt Design	power_opt_design
Place Design	place_design
Phys Opt Design	phys_opt_design
Route Design	route_design
Write Bitstream (all devices except Versal)	write_bitstream
Write Device Image (Versal devices)	write_device_image

For a complete description of the Tcl reporting commands and their options, see the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 19].

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 Vivado IP.

IMPORTANT: *NGC format files are not supported in the Vivado Design Suite for UltraScale™ devices. It is recommended that you regenerate the IP using the Vivado Design Suite IP customization tools with*

native output products. Alternatively, you can use the convert_ngc Tcl utility to convert NGC files to EDIF or Verilog formats. However, Xilinx recommends using native Vivado IP rather than XST-generated NGC format files going forward.

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. Processes constraints from XDC files.

   These constraints include both timing constraints and physical constraints such as package pin assignments and Pblocks for floorplanning.

IMPORTANT: *Review critical warnings that identify failed constraints. Constraints might be placed on design objects that have been optimized or no longer exist. The Tcl command 'write_xdc -constraints INVALID' also captures invalid XDC constraints.*

4. Builds placement macros.

   The Vivado tools create placement macros of cells, based on their connectivity or placement constraints to simplify placement.

   Examples of placement macros include:

   - An XDC-based macro.
   - 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 CLBs.

     Note: The primitives making up the carry chains must belong to a single macro to ensure that downstream placement aligns it into vertical slices.

Tcl Commands

The Tcl commands shown in Table 2-2 can be used to read the synthesized design into memory, depending on the source files in the design, and the state of the design.

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

Command	Project Mode	Non-Project Mode
`synth_design`	X	X
`open_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 Syntax

```
synth_design  [-name <arg>] [-part <arg>] [-constrset <arg>] [-top <arg>]
              [-include_dirs <args>] [-generic <args>] [-verilog_define <args>]
              [-flatten_hierarchy <arg>] [-gated_clock_conversion <arg>]
              [-directive <arg>] [-rtl] [-bufg <arg>] [-no_lc]
              [-shreg_min_size <arg>] [-mode <arg>] [-fsm_extraction <arg>]
              [-rtl_skip_mlo] [-rtl_skip_ip] [-rtl_skip_constraints]
              [-srl_style <arg>] [-keep_equivalent_registers]
              [-resource_sharing <arg>] [-cascade_dsp <arg>]
              [-control_set_opt_threshold <arg>] [-incremental <arg>]
              [-max_bram <arg>] [-max_uram <arg>] [-max_dsp <arg>]
              [-max_bram_cascade_height <arg>] [-max_uram_cascade_height <arg>]
              [-retiming] [-no_srlextract] [-assert] [-no_timing_driven]
              [-sfcu] [-debug_log] [-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) [Ref 21] and the *Vivado Design Suite User Guide: Synthesis* (UG901) [Ref 8].

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 tools into memory when `synth_design` completes. A design checkpoint is written after completing synthesis.

For more information on the `synth_design` Tcl command, see this link in the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 19]. This reference guide also provides a complete description of the Tcl commands and their options.

open_checkpoint

The open_checkpoint command opens a design checkpoint file (DCP), creates a new in-memory project and initializes a design immediately in the new project with the contents of the checkpoint. This command can be used to open a top-level design checkpoint, or the checkpoint created for an out-of-context module.

Note: In previous releases, the read_checkpoint command was used to read and initialize checkpoint designs. Beginning in version 2014.1, this function is provided by the open_checkpoint command.The behavior of read_checkpoint has been changed such that it only adds the checkpoint file to the list of source files. This is consistent with other read commands such as read_verilog, read_vhdl, and read_xdc. A separate link_design command is required to initialize the design and load it into memory when using read_checkpoint.

When opening a checkpoint, there is no need to create a project first. The open_checkpoint command reads the design data into memory, opening the design in Non-Project Mode. Refer to this link in the *Vivado Design Suite User Guide: Design Flows Overview* (UG892) [Ref 1] for more information on Project Mode and Non-Project Mode.

IMPORTANT: *In the incremental compile flow, the read_checkpoint command is still used to specify the reference design checkpoint.*

open_checkpoint Syntax

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

open_checkpoint Example Script

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

The open_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 tools.

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 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 tools memory from the completed synthesis run (also named `synth_1`).

If you use `open_run` while a design is already in memory, the Vivado tools prompt 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. Use `link_design -part <arg>` without a netlist loaded, to open a blank design for device exploration.*

link_design Syntax

```
link_design  [-name <arg>] [-part <arg>] [-constrset <arg>] [-top <arg>]
             [-mode <arg>] [-pr_config <arg>] [-reconfig_partitions <args>]
             [-partitions <args>] [-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 tools prompt 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 tools, 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.*

Note: For more information on the Partial Reconfiguration options of `link_design`, see this link in *Vivado Design Suite User Guide: Partial Reconfiguration* (UG909) [Ref 15].

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. For more detailed descriptions of the `report_timing_summary` command, see this link in the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 19].

BUFG Optimization

Mandatory logic optimization (MLO), which occurs at the beginning of `link design`, supports the use of the CLOCK_BUFFER_TYPE property to insert global clock buffers. Supported values are BUFG for 7 series, and BUFG and BUFGCE for UltraScale, UltraScale+, and Versal. The value NONE can be used for all architectures to suppress global clock buffer insertion through MLO and opt_design. For BUFG and BUFGCE, MLO inserts the corresponding buffer type to drive the specified net.

Use of CLOCK_BUFFER_TYPE provides the advantage of controlling buffer insertion using XDC constraints so that no design source or netlist modifications are required. Buffers inserted using CLOCK_BUFFER_TYPE are not subject to any limits, so the property must be used cautiously to avoid introducing too many global clocks into the design, which may result in placement failures. For more information, see the *Vivado Design Suite Properties Reference Guide* (UG912) [Ref 14].

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.

Often design connectivity errors are propagated to the logic optimization step where the flow fails. It is important to ensure valid connectivity using DRC Reports before running implementation.

Logic optimization skips optimization of cells and nets that have `DONT_TOUCH` properties set to a value of TRUE. Logic optimization also skips optimization of design objects that have directly applied timing constraints and exceptions. This prevents constraints from being lost when their target objects are optimized away from the design. An Info message at the end of each optimization stage provides a summary of the number of optimizations prevented due to constraints. Specific messages about which constraint prevented which optimizations can be generated with the `-debug_log` switch.

The Tcl command used to run Logic Optimization is `opt_design`.

Common Design Errors

One common error that can cause logic optimization to fail is using undriven LUT inputs, where the input is used by the LUT logic equation. This results in an error such as:

```
ERROR: [Opt 31-67] Problem: A LUT6 cell in the design is missing
a connection on input pin I0, which is used by the LUT equation.
```

This error often occurs when the connection was omitted while assembling logic from multiple sources. Logic optimization identifies both the cell name and the pin, so that it can be traced back to its source definition.

Available Logic Optimizations

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

IMPORTANT: *Logic optimization can be limited to specific optimizations by choosing the corresponding command options. Only those specified optimizations are run, while all others are disabled, even those normally performed by default.*

The following table describes the order in which the optimizations are performed when more than one option is selected. This ordering ensures that the most efficient optimization is performed.

Table 2-3: **Optimization Ordering for Multiple Options**

Phase	Name	Option	Default
1	Retargeting	`-retarget`	X
2	Constant Propagation	`-propconst`	X
3	Sweep	`-sweep`	X
4	Mux Optimization	`-muxf_remap`	
5	Carry Optimization	`-carry_remap`	
6	Control Set Merging	`-control_set_merge`	
7	Equivalent Driver Merging	`-merge_equivalent_drivers`	
8	BUFG Optimization	`-bufg_opt`	X
9	Shift Register Optimization	`-shift_register_opt`	X
10	MBUFG Optimization	`-mbufg_opt`	
11	DSP Register Opt	`-dsp_register_opt`	
12	Control Set Reduction	`(property controlled)`	X
13	Module-Based Fanout Opt	`-hier_fanout_limit <arg>`	
14	Remap	`-remap`	
15	Resynth Area	`-resynth_area`	

Table 2-3: **Optimization Ordering for Multiple Options *(Cont'd)***

Phase	Name	Option	Default
16	Resynth Sequential Area	`-resynth_seq_area`	
17	Block RAM Power Opt	`-bram_power_opt`	X

Phase 4 and 5 are not supported for Versal. Phase 10 is only supported for Versal.

When an optimization is performed on a primitive cell, the `OPT_MODIFIED` property of the cell is updated to reflect the optimizations performed on the cell. When multiple optimizations are performed on the same cell, the `OPT_MODIFIED` value contains a list of optimizations in the order they occurred. The following table lists the `OPT_MODIFIED` property value for the various `opt_design` options:

Table 2-4: **Optimization Options and Values**

opt_design Option	OPT_MODIFIED Value
-bufg_opt	BUFG_OPT
-carry_remap	CARRY_REMAP
-control_set_merge	CONTROL_SET_MERGE
-hier_fanout_limit	HIER_FANOUT_LIMIT
-merge_equivalent_drivers	MERGE_EQUIVALENT_DRIVERS
-muxf_remap	MUXF_REMAP
-propconst	PROPCONST
-remap	REMAP
-resynth_area	RESYNTH_AREA
-resynth_seq_area	RESYNTH_AREA
-retarget	RETARGET
-shift_register_opt	SHIFT_REGISTER_OPT
-sweep	SWEEP

Retargeting (Default)

When retargeting the design from one device family to another, retarget one type of block to another. For example, retarget instantiated MUXCY or XORCY components into a CARRY4 block; or retarget DCM to MMCM. In addition, simple cells such as inverters are absorbed into downstream logic. When the downstream logic cannot absorb the inverter, the inversion is pushed in front of the driver, eliminating the extra level of logic between the driver and its loads. After the transformation, the driver's INIT value is inverted and set/reset logic is transformed to ensure equivalent functionality.

Constant Propagation (Default)

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

- Eliminated logic:
  For example, an AND with a constant 0 input
- Reduced logic:
  For example, A 3-input AND with a constant 1 input is reduced to a 2-input AND.
- Redundant logic:
  For example, A 2-input OR with a logic 0 input is reduced to a wire.

Sweep (Default)

Sweep removes cells that have no loads.

Mux Optimization

Remaps MUXF7, MUXF8, and MUXF9 primitives to LUT3 to improve routability. You can limit the scope of mux remapping by using the `MUXF_REMAP` cell property instead of the `-muxf_remap` option. Set the `MUXF_REMAP` property to `TRUE` on individual MUXF primitives.

Note: Not applicable to Versal.

TIP: *To further optimize the netlist after the mux optimization is performed, combine the mux optimization with remap (`opt_design -muxf_remap -remap`).*

Carry Optimization

Remaps CARRY4 and CARRY8 primitives of carry chains to LUTs to improve routability. When running with the `-carry_remap` option, only single-stage carry chains are converted to LUTs. You can control the conversion of individual carry chains of any length by using the `CARRY_REMAP` cell property. The `CARRY_REMAP` property is an integer that specifies the maximum carry chain length to be mapped to LUTs. The `CARRY_REMAP` property is applied to CARRY4 and CARRY8 primitives and each CARRY primitive within a chain must have the same value to convert to LUTs. The minimum supported value is 1.

Example: A design contains multiple carry chains of lengths 1, 2, 3, and 4 CARRY8 primitives.

The following assigns a `CARRY_REMAP` property on all CARRY8 primitives:

```
Vivado% set_property CARRY_REMAP 2 [get_cells -hier -filter {ref_name == CARRY8}]
```

After `opt_design`, only carry chains of length 3 or greater CARRY8 primitives remain mapped to CARRY8. Chains with a length of 1 and 2 are mapped to LUTs.

Note: Not applicable to Versal.

TIP: *Remapping long carry chains to LUTs may significantly increase delay even with further optimization by adding the remap option. Xilinx recommends only remapping smaller carry chains, those consisting of one or two cascaded CARRY primitives.*

Control Set Merging

Reduces the drivers of logically-equivalent control signals to a single driver. This is like a reverse fanout replication, and results in nets that are better suited for module-based replication.

Equivalent Driver Merging

Reduces the drivers of all logically-equivalent signals to single drivers. This is similar to control set merging but is applied to all signals, not only control signals.

You can limit the scope of equivalent driver and control set merging by using the `EQUIVALENT_DRIVER_OPT` cell property. Setting the `EQUIVALENT_DRIVER_OPT` property to `MERGE` on the original driver and its replicas triggers the merge equivalent driver phase during `opt_design` and merges the drivers with that property. Setting the `EQUIVALENT_DRIVER_OPT` property to `KEEP` on the original driver and its replicas prevents the merging of the drivers with that property during the equivalent driver merging and the control set merging phase.

Note: Some interfaces require a one to one mapping from FF driver to interface pin and merging these logically-equivalent signals to a single driver can result in unroutable nets. In that case set a DONT_TOUCH property to TRUE or set the EQUIVALENT_DRIVER_OPT property to KEEP on those registers.

BUFG Optimization (Default)

Logic optimization conservatively inserts global clock buffers on clock nets and high-fanout non-clock nets such as device-wide resets. In Versal devices BUFG_FABRIC clock buffers are inserted on high-fanout non-clock nets.

For 7 series designs, clock buffers are inserted as long as 12 total global clock buffers are not exceeded.

For UltraScale, UltraScale+, and Versal designs, clock buffers are inserted as long as 24 total global clock buffers are not exceeded, not including BUFG_GT buffers.

For non-clock nets:

- The fanout must be above 25,000.
- The clock period of the logic driven by the net is below a device/speed grade specific limit.

For fabric-driven clock nets, the fanout must be 30 or greater.

Note: To prevent BUFG Optimization on a net, assign the value `NONE` to the `CLOCK_BUFFER_TYPE` property of the net. Some clock buffer insertion that is required to legalize the design can also occur in mandatory logic optimization.

MBUFG Optimization

For Versal devices, a new Multi-Clock Buffer (MBUFG) provides divide by 1, 2, 4, 8 clocks of the clock input on its O1, O2, O3, O4 outputs. The MBUFG clock outputs are all routed on the same global clock routing resources and only divided once they reach the BUFDIV_LEAF route-thru Bels. MBUFG driven clocks consume less routing resources and clock skew is minimized for synchronous CDC paths between clocks driven by the same MBUFG because the common node is closer to the source and destination.

The MBUFG optimization transforms parallel clock buffers driven by a common driver or clock modifying block (CMB), such as MMCM, DPLL, or XPLL, to MBUFG. The transformation occurs if the divide factors of the parallel clocks are divide by 1, 2, 4, 8 of a common clock. For CMB driven clocks, the phase shift has to be 0 and the duty cycle 50%. If the clock nets driven by the BUFGs have conflicting constraints such as CLOCK_DELAY_GROUP or USER_CLOCK_ROOT the transformation is also prevented. The transformation is only occurring when it is safe to do so without corrupting timing constraints. The following transformations are supported:

- Parallel BUFGCEs connected to a CMB to an MBUFGCE.
- Parallel BUFGCE_DIVs connected to a common clock driver to an MBUFGCE.
- Parallel BUFG_GTs connected to a common clock driver to an MBUFG_GT.

In addition to the global optimization using the -mbufg_opt option, you can control the conversion of selected BUFGs to MBUFG using the MBUFG_GROUP property. You must set the MBUFG_GROUP constraint on the net segment directly connected to the clock buffer. The following example shows the property applied to two clock nets, which are directly driven by the clock buffers:

```
set_property MBUFG_GROUP grp1 [get_nets -of [get_pins {BUFG_inst_0/O BUFG_inst_1/O}]
```

1. The picture in Figure 2-14 shows an MMCM driving several BUFGCE buffers. The CLKOUTn driven clocks are integer divides of 1, 2, 4, 8 of the CLKOUT1 driven clock. After the MBUFG transformation the four BUFGCEs are transformed to a single MBUFGCE and the CLKOUT1 driven clock is connected to the MBUFGCE I pin. The loads that were driven by the BUFGCEs are connected to the MBUFGCE O1, O2, O3, O4 pins.

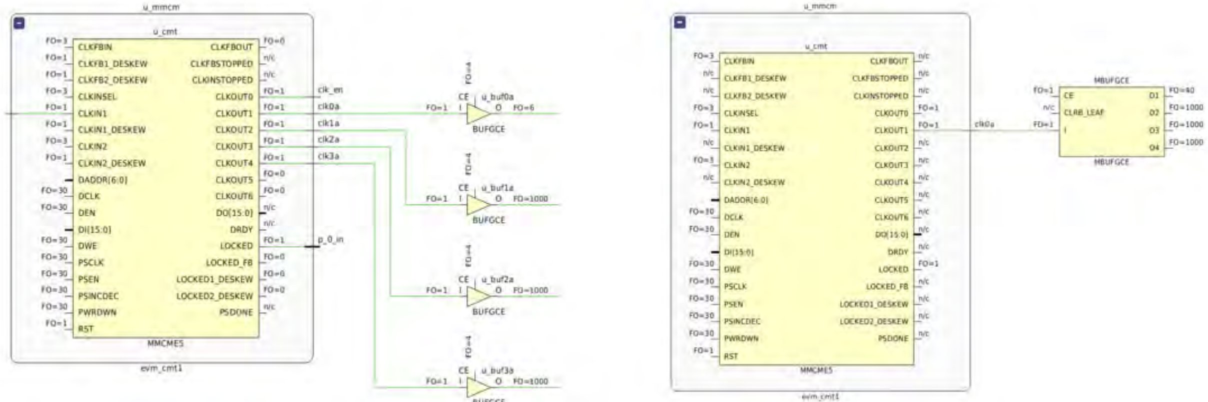


Figure 2-14: **MBUFG Optimization**

Shift Register Optimization (Default)

Shift register optimization includes multiple transformations.

- SRL fanout optimization: if an SRL (LUT-based shift register) primitive drives a fanout of 100 or greater, a register stage is taken from the end of the SRL chain and transformed into a register primitive. This enables more flexible downstream replication if the net becomes timing-critical. In general it is easier to replicate high-fanout register drivers compared to high fanout SRL drivers.
- Transformation between SRL and register primitives:
    - An SRL primitive can be converted to a logically equivalent chain of register primitives using the `SRL_TO_REG` property with a value of true. This transform is typically used to increase the number of available pipeline register stages that can be spread to allow signals to traverse long distances within a device. Increasing the number of register stages can increase the clock frequency at the expense of higher latency.
    - A chain of register primitives can be converted to a logically equivalent SRL primitive using the `REG_TO_SRL` property with a value of true. This transform is typically used to reduce the number of pipeline register stages used by signals to traverse long distances within a device. Having too many register stages may create congestion or other placement problems.
- Selective movement of pipeline stages between SRLs and register chains: These transformations can be used when a pipeline register chain consists of SRLs and register primitives. A register stage can be pulled out of or pushed into SRLs on either the SRL inputs or SRL outputs. This allows increased control of pipeline register structures to address under and over-pipelining.

- Under-pipelining: To pull a register from an SRL through its input, apply the SRL property `SRL_STAGES_TO_REG_INPUT` with value 1. To pull a register stage from an SRL output, apply `SRL_STAGES_TO_REG_OUTPUT` with value 1.
- Over-pipelining: To push a register into an SRL input, apply the SRL property `SRL_STAGES_TO_REG_INPUT` with value -1. To push a register stage into an SRL output, apply `SRL_STAGES_TO_REG_OUTPUT` with value -1.

Note: All transforms from registers to SRLs are only possible if control sets are compatible.

Shift Register Remap

This is a set of optimizations that convert shift registers between discrete register chains and SRLs which are the LUTRAM-based shift register primitives. These optimizations specify global thresholds to convert from one form to another. The optimizations are used to balance utilization of registers and LUTRAM-based SRLs. High SRL utilization can lead to congestion and converting small SRLs to registers can help ease congestion and simultaneously improve performance by providing discrete registers to cover more distance for critical paths. However congestion can emerge again when register utilization becomes too high. Converting very long register chains to SRLs can absorb register stages and their routing which helps reduce congestion.

The optimizations are accessed using the `-srl_remap_mode` option which takes a Tcl list of lists as an argument to define the mode. Following are the different types of optimizations.

- Converting small SRLs to registers: For this optimization use the `max_depth_srl_to_ffs` mode:
  - `opt_design -srl_remap_modes {{max_depth_srl_to_ffs <depth>}}`
  - Here all SRLs of `depth <depth>` and smaller are remapped to register chains.
- Converting large shift register chains to SRLs: For this optimization use the `min_depth_ffs_to_srl` mode:
  - `opt_design -srl_remap_modes {{min_depth_ffs_to_srl <depth>}}`
  - Here all register chains greater than `depth <depth>` are remapped to SRL primitives.
- Automatic target utilization optimizations: This mode uses the following syntax:
  - `-srl_remap_modes {{target_ff_util <ff_util> target_lutram_util <lutram_util>}}`

Here you specify percent utilization targets (0 to 100) for both registers and LUTRAMs. If the current utilization exceeds a target, Vivado will convert from the overutilized resource type to the other until the utilization target is met. When converting from SRLs to registers, Vivado begins with the smallest SRLs. When converting from registers to SRLs, Vivado begins with the largest register chains.

Note: The max_depth_srl_to_ffs and min_depth_ffs_to_srl can be used simultaneously but cannot be used with the target utilization settings.

DSP Register Opt

This option is used to perform various optimizations on DSP slice pipeline, input and output registers to improve timing within and to and from the DSP slices. The table below lists the available optimization.

Note: Not applicable to Versal.

Table 2-5: **DSP Register Opt Available Optimizations**

Optimization Type	Configuration Required to Trigger	Post Optimization State	Timing Requirement
MREG to PREG	MREG=1, PREG=0	MREG=0, PREG=1	Timing from MREG is critical (slack less than 0.5ns), and timing to MREG is not critical (slack greater than 1ns)
PREG to MREG	MREG=0, PREG=1	MREG=1, PREG=0	Timing to PREG is critical (slack less than 0.5ns), and timing from PREG is not critical (slack greater than 1ns)
MREG to ADREG	ADREG=0, MREG=1	ADREG=1, MREG=0	Timing to MREG is critical (slack less than 0.5ns), and timing from MREG is not critical (slack greater than 1ns)
ADREG to MREG	ADREG=1, MREG=0	ADREG=0, MREG=1	Timing from ADREG is critical (slack less than 0.5ns), and timing to ADREG is not critical (slack greater than 1ns)
AREG/BREG push out to fabric	AREG=1/2, BREG=1/2	AREG=0/1, BREG=0/1, FDRE in fabric	Timing to AREG/BREG is critical (slack less than 0.5ns), and timing from AREG/BREG is not critical (slack greater than 1ns)
AREG/BREG pull in from fabric	AREG=0/1, BREG=0/1, FDRE in fabric	AREG=1/2, BREG=1/2	Timing to DSP input is critical (slack less than 0.5ns)
AREG and BREG to MREG	AREG=1/2, BREG=1/2, MREG=0	AREG=0/1, BREG=0/1, MREG=1	Timing from AREG/BREG is critical (slack less than 0.5ns), and timing to AREG/BREG is not critical (slack greater than 1ns)

Table 2-5: **DSP Register Opt Available Optimizations**

Optimization Type	Configuration Required to Trigger	Post Optimization State	Timing Requirement
MREG to AREG and BREG	AREG=0, BREG=0, MREG=1	AREG=1, BREG=1, MREG=0	Timing to MREG is critical (slack less than 0.5ns), and timing from MREG is not critical (slack greater than 1ns)
PREG push out to fabric	PREG=1	PREG=0, FDRE in fabric	Timing from PREG is critical (slack less than 0.5ns), and timing to PREG is not critical (slack greater than 1ns)
PREG pull in from fabric	PREG=0, FDRE in fabric	PREG=1	Timing from DSP output is critical (slack less than 0.5ns)

Control Set Reduction

Designs with several unique control sets can have fewer options for placement, resulting in higher power and lower performance. Designs with fewer control sets have more options and flexibility in terms of placement, generally resulting in improved results. The number of unique control sets can be reduced by applying the `CONTROL_SET_REMAP` property to a register that has a control signal driving the synchronous set/reset pin or CE pin. This triggers the optional control set reduction phase and maps the set/reset and/or CE logic to the D-input of the register. If possible, the logic is combined with an existing LUT driving the D-input, which prevents extra levels of logic.

The `CONTROL_SET_REMAP` property supports the following values:

- `ENABLE` - Remaps the EN input to the D-input.
- `RESET` - Remaps the synchronous S or R input to the D-input.
- `ALL` - Same as `ENABLE` and `RESET`.
- `NONE` or unset - No optimization (Default).

Note: This optimization is automatically triggered when the `CONTROL_SET_REMAP` property is detected on any register.

Module-Based Fanout Optimization

Net drivers with fanout greater than the specified limit, provided as an argument with this option, will be replicated according to the logical hierarchy.

For each hierarchical instance driven by the high-fanout net, if the fanout within the hierarchy is greater than the specified limit, then the net within the hierarchy is driven by a replica of the driver of the high-fanout net.

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

Remap

Remap combines multiple LUTs into a single LUT to reduce the depth of the logic. Selective remap can be triggered by applying the `LUT_REMAP` property to a group of LUTs. Chains of LUTs with `LUT_REMAP` values of `TRUE` are collapsed into fewer logic levels where possible. Remap optimization can combine LUTs that belong to different levels of logical hierarchy into a single LUT to reduce logic levels. Remapped logic is combined into the LUT that is furthest downstream in the logic cone.

This optimization also replicates LUTs with the `LUT_REMAP` property that have fanout greater than one before the transformation.

Note: Setting the `LUT_REMAP` property to `FALSE` does not prevent LUTs from getting remapped when running `opt_design` with the `-remap` option.

Aggressive Remap

Similar to Remap, Aggressive Remap combines multiple LUTs into a single LUT to reduce logic depth. Aggressive Remap is a more exhaustive optimization than Remap, and may achieve further logic level reduction than Remap at the expense of longer runtime.

Resynth Area

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

Resynth Sequential Area

Resynth Sequential Area performs re-synthesis to reduce both combinational and sequential logic. Performs a superset of the optimization of Resynth Area.

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.

Property-Only Optimization

This is a non-default option where `opt_design` runs only those phases that are triggered by opt_design properties. If no such properties are found, opt_design exits and leaves the design unchanged.

Following are opt_design cell properties that trigger optimizations when using this option:

- `MUXF_REMAP`
- `CARRY_REMAP`
- `SRL_TO_REG`
- `REG_TO_SRL`
- `SRL_STAGES_TO_REG_INPUT`
- `SRL_STAGES_TO_REG_OUTPUT`
- `LUT_REMAP`
- `CONTROL_SET_REMAP`
- `EQUIVALENT_DRIVER_OPT`

opt_design

The `opt_design` command runs Logic Optimization.

opt_design Syntax

```
opt_design  [-retarget] [-propconst] [-sweep] [-bram_power_opt] [-remap]
            [-aggressive_remap] [-resynth_area] [-resynth_seq_area]
            [-directive <arg>] [-muxf_remap] [-hier_fanout_limit <arg>]
            [-bufg_opt] [-mbufg_opt] [-shift_register_opt] [-dsp_register_opt]
            [-srl_remap_modes <arg>] [-control_set_merge]
            [-merge_equivalent_drivers] [-carry_remap] [-debug_log]
            [-property_opt_only] [-quiet] [-verbose]
```

opt_design Example Script

```
# Run logic optimization with the remap optimization enabled, save results in a
checkpoint, report timing estimates
opt_design -directive AddRemap
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, the following is another method for skipping the block RAM optimization that is run by default:

```
opt_design -retarget -propconst -sweep -bufg_opt -shift_register_opt
```

Using Directives

Directives provide different modes of behavior for the `opt_design` command. Only one directive can 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 combinational logic.
- `AddRemap`:
  Runs the default logic optimization flow and includes LUT remapping to reduce logic levels.
- `ExploreSequentialArea`:
  Runs multiple passes of optimization with emphasis on reducing registers and related combinational logic.
- `RuntimeOptimized`:
  Runs minimal passes of optimization, trading design performance for faster run time.
- `NoBramPowerOpt`:
  Runs all the default `opt_design` optimizations except block RAM Power Optimization.
- `ExploreWithRemap`:
  Same as the `Explore` directive but includes the `Remap` optimization.
- `Default`:
  Runs `opt_design` with default settings.

Table 2-6 provides an overview of the optimization phase for the different directives.

Table 2-6: **Optimization Phases for Directives**

Phase	Default	Explore	ExploreWithRemap	ExploreArea	AddRemap	Explore SequentialArea	NoBramPowerOpt	RuntimeOptimized
1	Retargeting	Retargeting	Retargeting	Retargeting	Retargeting	Retargeting	Retargeting	Retargeting
2	Constant Propagation	Constant Propagation	Constant Propagation	Constant Propagation	Constant Propagation	Constant Propagation	Constant Propagation	Constant Propagation
3	Sweep	Sweep	Sweep	Sweep	Sweep	Sweep	Sweep	Sweep
4	BUFG Optimization	BUFG Optimization	BUFG Optimization	BUFG Optimization	BUFG Optimization	BUFG Optimization	BUFG Optimization	BUFG Optimization
5	Shift Register Optimization	Shift Register Optimization	Shift Register Optimization	Shift Register Optimization	Shift Register Optimization	Shift Register Optimization	Shift Register Optimization	Shift Register Optimization
6	Block RAM Power Opt	Constant Propagation	Constant Propagation	Constant Propagation	Remap	Constant Propagation		
7		Sweep	Sweep	Sweep	Block RAM Power Opt	Sweep		
8		Block RAM Power Opt[a]	Remap	Resynth Area		Resynth Area		
9			Block RAM Power Opt[a]	Block RAM Power Opt		Resynth Sequential Area		
10						Block RAM Power Opt		

a. Phase not run in UltraScale/UltraScale+/Versal designs.

Using the -debug_log and -verbose Options

To better analyze optimization results, use the `-debug_log` option to see additional details of the logic affected by `opt_design` optimization. The log displays additional messages of logic that is reduced due to constant values and loadless logic that is subject to removal. The log also displays detailed messages about optimizations that are prevented due to constraints.

Use the `-verbose` option to see full details of all logic optimization performed by `opt_design`. 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.

RECOMMENDED: *To improve tool run time for large designs, use the `-verbose` option only in shell or batch mode and not in the GUI mode.*

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. Therefore you must reload the synthesized design before adding either the `-debug_log` or `-verbose` options.*

Logic Optimization Constraints

Logic Preservation

The Vivado Design Suite respects the `DONT_TOUCH` property during logic optimization. It does not optimize away nets or cells with these properties. To speed up the net selection process, nets with DONT_TOUCH properties are pre-filtered and not considered for physical optimization. For more information, see this link in the *Vivado Design Suite User Guide: Synthesis* (UG901) [Ref 8].

You would typically apply the `DONT_TOUCH` property to leaf cells to prevent them from being optimized. `DONT_TOUCH` on a hierarchical cell preserves the cell boundary, but optimization might still occur within the cell and constants can still be propagated across the boundary. To preserve a hierarchical net, apply the `DONT_TOUCH` property to all net segments using the `-segments` option of `get_nets`.

The tools automatically add `DONT_TOUCH` properties of value `TRUE` to nets that have `MARK_DEBUG` properties of value `TRUE`. This is done to keep the nets intact throughout the implementation flow so that they can be probed at any design stage. This is the recommended use of `MARK_DEBUG`. However, on rare occasions `DONT_TOUCH` might be too restrictive and could prevent optimization such as constant propagation, sweep, or remap, leading to more difficult timing closure. In such cases, you can set `DONT_TOUCH` to a value of `FALSE`, while keeping `MARK_DEBUG` `TRUE`. The risk in doing this is that nets with `MARK_DEBUG` can be optimized away and no longer probed.

Logic Optimization

Certain optimizations can be performed on specific objects rather than the entire design. These optimizations are triggered by object properties. Logic Optimization detects the presence of these properties and automatically runs the necessary optimization phases. This is true for all properties except for shift register optimizations properties, which require the `-shift_register_opt` option. The following is a summary of properties for object-specific optimization.

Table 2-7: **Logic Optimization Properties**

Property	Description
MUXF_REMAP	Set to TRUE on MUXF primitives to convert them to LUTs
CARRY_REMAP	Set the threshold on CARRY primitives to convert to LUTs
SRL_TO_REG[1]	Set to TRUE on SRL primitives to convert them to register chains
REG_TO_SRL[1]	Set to TRUE on register chains to convert them to SRL primitives
SRL_STAGES_TO_REG_INPUT[1]	Set to the appropriate value on an SRL primitive to move a register across its input
SRL_STAGES_TO_REG_OUTPUT[1]	Set to the appropriate value on an SRL primitive to move a register across its output
LUT_REMAP	Set to TRUE on cascaded LUTs to reduce LUT levels
CONTROL_SET_REMAP	Set on registers to specify the type of control signal to remap to LUTs
EQUIVALENT_DRIVER_OPT	Set on logically-equivalent drivers to force or prevent merging

Notes:

1. Requires `-shift_register_opt` option to perform optimization.

Power Optimization

Power optimization is an optional step that optimizes dynamic power using clock gating. 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 your design, without altering functionality.

For more information, see the *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907) [Ref 11].

Vivado Tools Power Optimization

The Vivado power optimization analyzes all portions of the design, including legacy and third-party IP blocks. It also identifies opportunities where actively changing signals can be clock-gated because they are not being read every clock cycle. This reduces switching activity which in turn reduces dynamic power.

Using Clock Enables (CEs)

The Vivado power optimizer takes advantage of the abundant supply of Clock Enables (CEs). Power optimization creates gating logic to drive register clock enables such that registers only capture data on relevant clock cycles.

Note that in actual silicon, 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

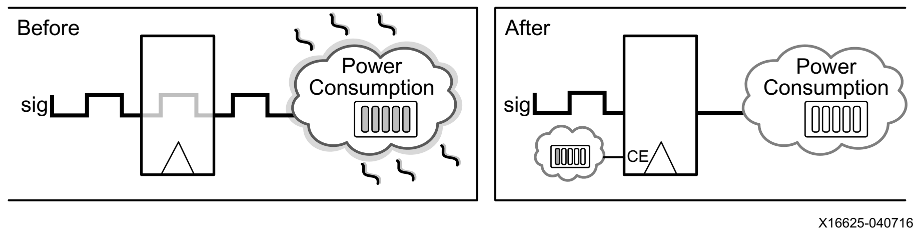


Figure 2-15: **Intelligent Clock Gating**

Intelligent clock gating also reduces power for dedicated block RAMs in either simple dual-port or true dual-port mode, as shown in Figure 2-16.

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.

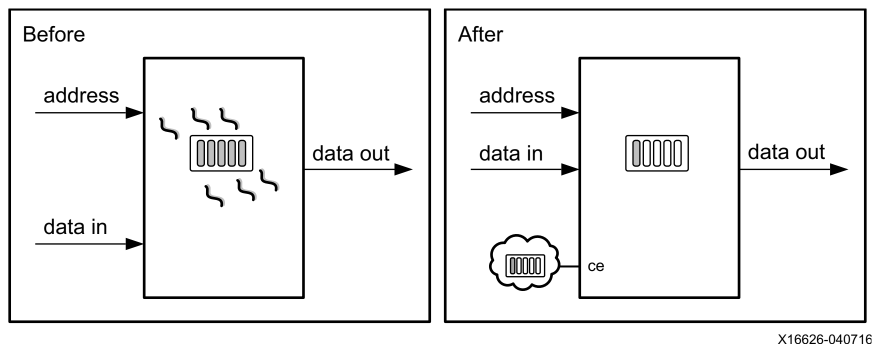


Figure 2-16: **Leveraging Block RAM Enables**

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. You can also use `set_power_opt` to specify the specific Block RAM cells for optimization in `opt_design`.

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`.

RECOMMENDED: *If you want to prevent block RAM Power Optimization on specific block RAMs during* `opt_design`*, use* `set_power_opt -exclude_cells [get_cells <bram_insts>]`*.*

Placement

The Vivado Design Suite placer places 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 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. Internal DRCs check for illegal placement, such as Memory IP 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. For UltraScale, UltraScale+, and Versal devices, the placer also assigns clock tracks and pre-routes the clocks. Register cells with IOB properties are processed during this phase to determine which registers with an IOB value of TRUE should be mapped to I/O logic sites. If the placer fails to honor an IOB property of TRUE, a critical warning is issued.

Placer Targets

The placer targets at this stage of placement are:

- I/O ports and their related logic
- Global 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 Memory IP 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 User Guide* (UG472) [Ref 17], *UltraScale Architecture Clocking Resources User Guide* (UG572) [Ref 18] and *Versal ACAP Clocking Resources Architecture Manual* (AM003)[Ref 16]. For example, an input that drives a global clock buffer must be located at

a clock-capable I/O site, must be located in the same upper or lower half of the device for 7 series devices, and in the same clock region for UltraScale devices. 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.

Placement can fail because of several reasons, including:

- Clock tree issues caused by conflicting constraints
- Clock tree issues that are too complex for the placer to resolve
- RAM and DSP block placement conflicts with other constraints, such as Pblocks
- Over-utilization of resources
- I/O bank requirements and rules

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 might 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.*

For more information about UltraScale clock tree placement and routing, see the *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949) [Ref 13].

Global Placement, Detailed Placement, and Post-Placement Optimization

After Clock and I/O placement, the remaining placement phases consist of global placement, detailed placement, and post-placement optimization.

Global Placement

Global placement consists of two major phases: floorplanning and physical synthesis.

Floorplanning Phase

During floorplanning, the design is partitioned into clusters of related logic and initial locations are chosen based on placement of I/O and clocking resources. When targeting SSI devices, the design is also partitioned into different SLRs to minimize SLR crossings and

their associated delay penalties. Soft SLR floorplan constraints can be applied to guide the logic partitioning during this phase. For more information about Using Soft SLR Floorplan Constraints, see the *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949) [Ref 13].

Physical Synthesis Phase

During physical synthesis, the placer can perform various physical optimizations that will optimize the netlist for later placement phases based on the initial placement of the design after the floorplanning stage. For example, for fanout based replication the replicated driver can be co-located with its loads because the initial placement is known. This alleviates congestion that can be introduced when replication is done without knowledge of placement prior to place_design. Optimizations are considered based on internal parameters and for timing based optimizations the timing is evaluated and the optimization is committed if timing is improved. The following optimizations are available as shown in as shown in Figure 2-16.

Summary of Physical Synthesis Optimizations

Optimization	Added Cells	Removed Cells	Optimized Cells/Nets	Dont Touch	Iterations	Elapsed
LUT Combining	173	4244	4417	0	1	00:00:10
Very High Fanout	193	0	19	0	1	00:00:11
Fanout	0	0	0	0	1	00:00:01
Critical Cell	0	0	0	0	1	00:00:00
DSP Register	0	0	0	0	1	00:00:01
Shift Register to Pipeline	0	0	0	0	1	00:00:00
Shift Register	5	0	1	0	1	00:00:01
BRAM Register	1040	0	15	0	1	00:00:46
URAM Register	0	0	0	0	1	00:00:01
Dynamic/Static Region Interface Net Replication	0	0	0	0	1	00:00:00
Critical Cell	0	0	0	0	1	00:00:00
Total	1411	4244	4452	0	11	00:01:10

Figure 2-16: **Summary of Physical Sythesis Optimizations**

- **LUT Decomposition and Combining**

  LUT Decomposition breaks LUT shapes if it improves timing (only LUTs with SOFT_HLUTNM property are considered). LUT combining combines LUTs if it improves utilization.

- **Very High-Fanout Optimization**

  Very High-Fanout Optimization replicates registers driving high-fanout nets (Fanout > 1000, Slack < 2.0 ns).

- **Critical Cell Optimization**

  Critical-Cell Optimization replicates cells in failing paths. If the loads on a specific cell are placed far apart, the cell might be replicated with new drivers placed closer to load clusters. This optimizations often applies to nets driving large BRAM or URAM arrays or large number of DSPs as the sites for these blocks are spread over a wider area of the device. High fanout is not a requirement for this optimization to occur (Slack < 0.5 ns).

- **Fanout Optimization**

Nets with a MAX_FANOUT property value that is less than the actual fanout of the net are considered for fanout optimization. The user can force the replication of a register or a LUT driving a net by adding the FORCE_MAX_FANOUT property to the net. The value of the FORCE_MAX_FANOUT specifies the maximum physical fanout the nets should have after the replication optimization. The physical fanout in this case refers to the actual site pin loads, not the logical loads. For example if the replica drives multiple LUTRAM loads that are all grouped in the same slice, the combined fanout will be 1 for all of the LUTRAMs in the same slice. The FORCE_MAX_FANOUT forces the replication during physical synthesis regardless of the slack of the signal. The user can force replication based on physical device attributes with the MAX_FANOUT_MODE property. The property can take on the value of CLOCK_REGION, SLR, or MACRO. For example, the MAX_FANOUT_MODE property with a value of CLOCK_REGION replicates the driver based on the physical clock region, the loads placed into same clock region will be clustered together. The MAX_FANOUT_MODE property takes precedence over the FORCE_MAX_FANOUT property and physical synthesis will try to honor both by applying MAX_FANOUT_MODE based optimization first and then all its replicated drivers will inherit the FORCE_MAX_FANOUT property to do further replication within a clock region. This is illustrated in the Figure 2-17 example where a register drives four loads; two registers and two MACRO loads (Block RAM, UltraRAM or DSP). Replication provides separate drivers for the register loads and MACRO loads and then the driver for the MACRO loads is replicated until the FORCE_MAX_FANOUT property value is satisfied.

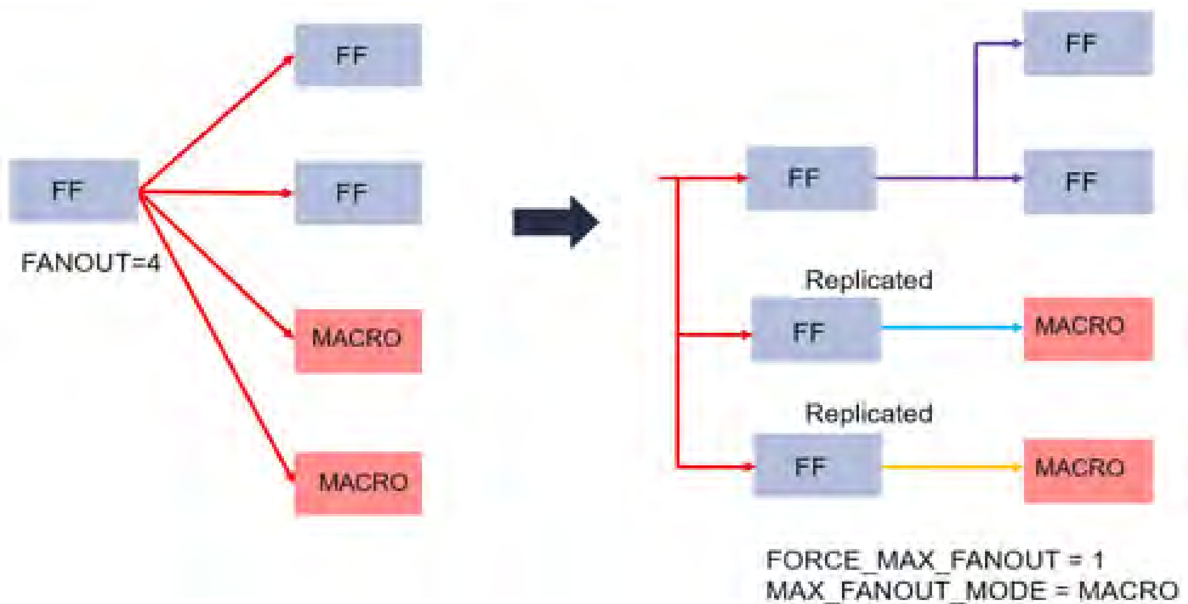


Figure 2-17: **Applying MAX_FANOUT_MODE with value MACRO together with FORCE_MAX_FANOUT**

- ***DSP Register Optimization***

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.

- **Shift Register to Pipeline Optimization**

  Shift Register to Pipeline Optimization turns a shift register with fixed length to dynamically adjusted register pipeline and places the pipeline optimally to improve timing. Only SRLs with the PHYS_SRL2PIPELINE attribute set to TRUE are considered for this optimization. The pull/push of FFs happens on the SRL's Q-pin. The SRL length needs to be fixed and dynamic SRLs are not supported for this optimization.

- **Shift Register Optimization**

  The shift register optimization improves timing on negative slack paths between shift register cells (SRLs) and other logic cells.

- **Block RAM Register Optimization**

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

- **URAM Register Optimization**

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

- **Dynamic/Static Region Interface Net Replication**

  Optimization to replicate drivers on static design to reconfigurable module boundary paths in DFX flow.

- **Equivalent Driver Rewire Optimization**

  This optimization redistributes loads between logically-equivalent drivers to minimize routing overlap and provide a more optimal co-location of drivers and loads. This helps reduce utilization and congestion and allows later placer stages to move drivers and loads more optimally to improve QoR. For more information on these optimizations see Available Physical Optimizations in the Physical Optimization section. Physical synthesis in the placer is run by default in all of the placer directives. At the end of the physical synthesis phase, a table shows the summary of optimizations.

Detailed Placement

Detailed placement takes the design from the initial global placement to a fully-placed design, generally starting with the largest structures (which serve as good anchors) down to the smallest. The detail placement process begins by placing large macros such as multi-column URAM, BRAM, and DSP block arrays, followed by LUTRAM array macros, and smaller macros such as user-defined XDC Macros. Logic placement is iterated to optimize

wirelength, timing, and congestion. LUT-FF pairs are packed into CLBs with the additional constraints that registers in the CLB must share common control sets.

Post-Placement Optimization

After all logic locations have been assigned, Post-Placement Optimization performs the final steps to improve timing and congestion. These include improving critical path placement and the optional BUFG insertion phase during which the placer can route high fanout nets on global routing tracks to free up fabric routing resources. High-fanout nets (fanout > 1000) driving control signals with a slack greater than 1.0ns are considered for this optimization. The loads are split between critical loads and high positive slack loads. The high positive slack loads are driven through a BUFGCE which is placed at the nearest available site to the original driver, whereas the critical loads remain connected to the original driver. This optimization is performed only if there is no timing degradation. BUFG Insertion is on by default and can be disabled with the `-no_bufg_opt` option.

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

place_design

The `place_design` command runs placement on the design. Like the other implementation commands, `place_design` is re-entrant 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] [-timing_summary]
             [-unplace] [-post_place_opt] [-no_psip] [-no_bufg_opt]
             [-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 can be specified at a time. The directive option is incompatible with other options

with the exception of -no_fanout_opt, -no_bufg_opt, -quiet, and -verbose. Use the -directive option to explore different placement options for your design.

Placer Directives

Because placement typically has the greatest impact on overall design performance, the Placer has the most directives of all commands. Table 2-8 shows which directives might benefit which types of designs.

Table 2-8: **Directive Guidelines**

Directive	Designs Benefitted
BlockPlacement	Designs with many block RAM, DSP blocks, or both
ExtraNetDelay	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
ExtraPostPlacementOpt	All design types
SSI	SSI designs that might 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. This directive can improve timing to and from RAM and DSP blocks.
- EarlyBlockPlacement:
  Timing-driven placement of RAM and DSP blocks. The RAM and DSP block locations are finalized early in the placement process and are used as anchors to place the remaining logic.
- ExtraNetDelay_high:
  Increases estimated delay of high fanout and long-distance nets. This directive can improve timing of critical paths that meet timing after place_design but fail timing in route_design due to overly optimistic estimated delays. Two levels of pessimism are supported: high and low. ExtraNetDelay_high applies the highest level of pessimism.
- ExtraNetDelay_low:
  Increases estimated delay of high fanout and long-distance nets. This directive can improve timing of critical paths that have met timing after place_design but fail timing in route_design due to overly optimistic estimated delays. Two levels of

pessimism are supported: high and low. `ExtraNetDelay_low` applies the lowest level of pessimism.

- `SSI_SpreadLogic_high`:
  Spreads logic throughout the SSI device to avoid creating congested regions. Two levels are supported: high and low. `SpreadLogic_high` achieves the highest level of spreading.
- `SSI_SpreadLogic_low`:
  Spreads logic throughout the SSI device to avoid creating congested regions. Two levels are supported: high and low. `SpreadLogic_low` achieves a minimal level of spreading.
- `AltSpreadLogic_high`:
  Spreads logic throughout the device to avoid creating congested regions. Three levels are supported: high, medium, and low. `AltSpreadLogic_high` achieves the highest level of spreading.
- `AltSpreadLogic_medium`:
  Spreads logic throughout the device to avoid creating congested regions. Three levels are supported: high, medium, and low. `AltSpreadLogic_medium` achieves a nominal level of spreading.
- `AltSpreadLogic_low`:
  Spreads logic throughout the device to avoid creating congested regions. Three levels are supported: high, medium, and low. `AltSpreadLogic_low` achieves a minimal level of spreading.
- `ExtraPostPlacementOpt`:
  Higher placer effort in post-placement optimization.
- `ExtraTimingOpt`:
  Use an alternate set of algorithms for timing-driven placement during the later stages.
- `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 run time.
- `Quick`:
  Absolute, fastest run time, non-timing-driven, performs the minimum required for a legal design.

- `Default`:
  Run `place_design` with default settings.

Using the -unplace Option

The `-unplace` option unplaces all cells and all ports in a design that do not have fixed locations. An object with fixed location has an `IS_LOC_FIXED` property value of TRUE.

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 -timing_summary Option

After placement, an estimated timing summary is output to the log file. By default, the number reflects the internal estimates of the placer. For example:

```
INFO: [Place 30-746] Post Placement Timing Summary WNS=0.022. For the most accurate
timing information please run report_timing.
```

For greater accuracy at the expense of slightly longer run time, you can use the `-timing_summary` option to force the placer to report the timing summary based on the results from the static timing engine.

```
INFO: [Place 30-100] Post Placement Timing Summary | WNS=0.236  | TNS=0.000  |
```

where

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

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.

Using the -post_place_opt Option

Post placement optimization is a placement optimization that can potentially improve critical path timing at the expense of additional run time. The optimization is performed on a fully placed design with timing violations. For each of the top few critical paths, the placer tries moving critical cells to improve delay and commits new cell placements if they improve

estimated delay. For designs with longer run times and relatively more critical paths, these placement passes might further improve timing.

Using the no_psip Option

The no_psip option disables the default physical synthesis algorithm in the placer.

Using the -no_bufg_opt Option

The -no_bufg_opt option disables the default BUFG insertion algorithm in the placer.

Auto-Pipelining

You can optionally insert additional pipeline registers during placement to address timing closure challenges on specific buses and interfaces.

Using the AXI Register Slice in Auto-Pipelining Mode

The AXI Register Slice IP core is typically used for adding pipeline registers between memory mapped or streaming AXI interfaces to help close timing. For larger devices, adding the right amount of pipelining without overly increasing the register utilization and the application latency is a common challenge. To simplify the pipeline insertion task and allow the Vivado placer more flexibility, you can use the auto-pipeline optimization feature for the AXI Register Slice IP core. When this feature is enabled, a special physical synthesis phase (between the floorplanning and global placer phases) inserts and places the additional pipeline stages based on setup timing slack and SLR distance. The AXI Register Slice IP core remains compliant with the AXI handshake protocol despite the increased latency due to the use of small FIFOs.

You can enable this feature in the IP Configuration Wizard. Set the Register Slice Options (REG_*) to **Multi SLR Crossing**. In addition, set the Use timing-driven pipeline insertion for all Multi-SLR channels option to **1** to enable auto-pipelining. The following figure shows an example.

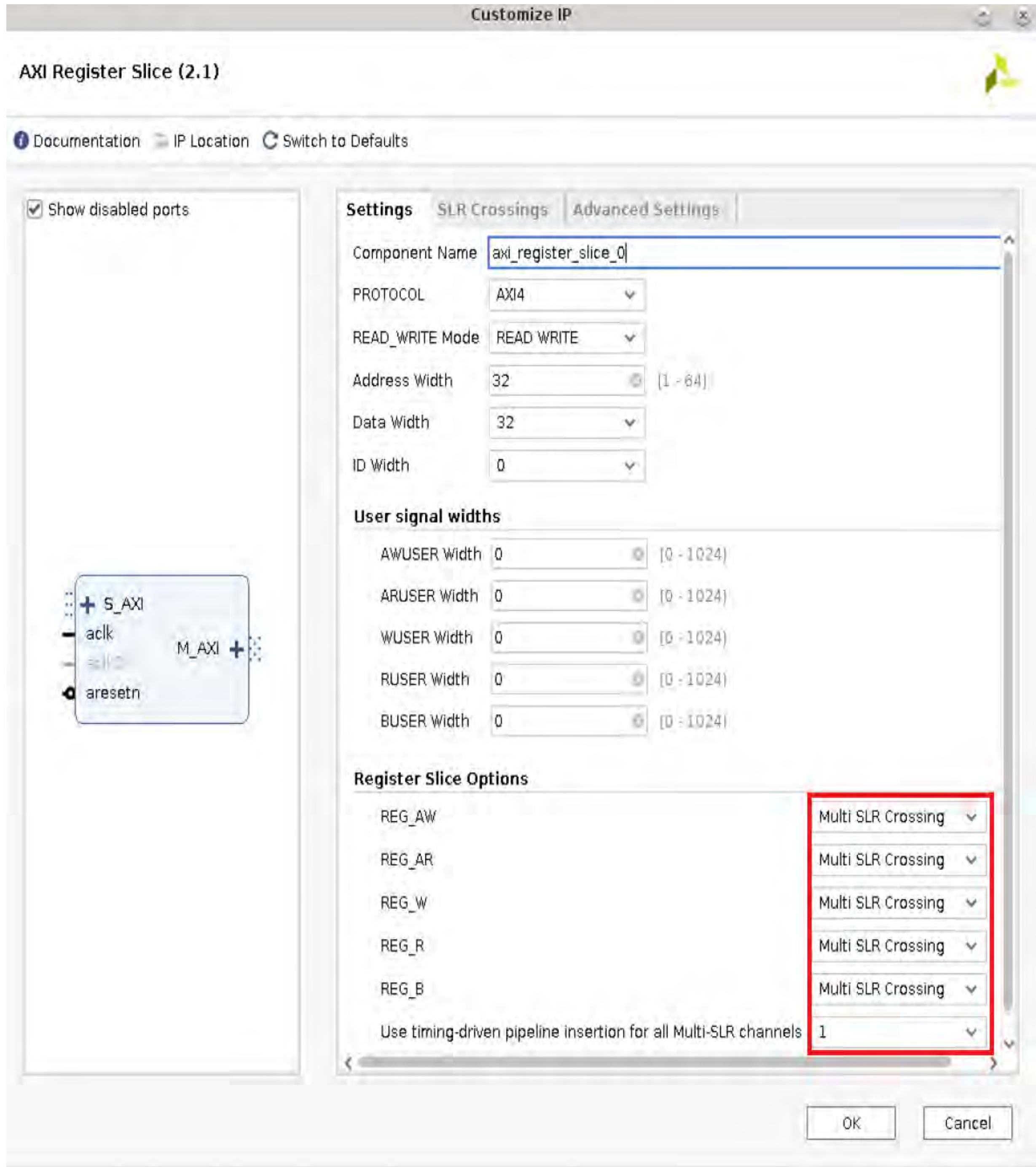


Figure 2-18: **Example AXI Register Slice IP Settings to Enable Auto-Pipelining Feature**

Using Auto-Pipelining on Custom Interfaces

Auto-pipelining is not limited to the AXI Register Slice IP. You can also control auto-pipelining on custom interfaces using the properties shown in the following table, which are specified in the RTL. For more information, see the *Vivado Design Suite Properties Reference Guide* (UG912) [Ref 14].

Table 2-9:

Property Name	Object	Format/Range	Description
AUTOPIPELINE_MODULE	hierarchical cell	Boolean	Establishes a separate name-space for all group names defined throughout sub-hierarchies. This property must be used when a module with auto-pipelining properties is instantiated several times in the design.
AUTOPIPELINE_GROUP	net	String (case-insensitive)	Establishes the auto-pipeline group name of signals that must receive an equal number of auto-inserted pipeline flip-flops.
AUTOPIPELINE_INCLUDE	net	String (case-insensitive)	Specifies the name of another AUTOPIPELINE_GROUP to include when applying the AUTOPIPELINE_LIMIT.
AUTOPIPELINE_LIMIT	net	0 < integer <= 24	Defines the maximum number of auto-inserted pipeline flip-flops for associated groups.

All nets that belong to the same AUTOPIPELINE_GROUP must have an equal number of pipeline registers inserted on each tagged signal. Following are additional considerations:

- If an AUTOPIPELINE_GROUP does not reference an AUTOPIPELINE_INCLUDE group, the number of pipeline stages inserted into the AUTOPIPELINE_GROUP must be between 0 and the AUTOPIPELINE_LIMIT.
- If an AUTOPIPELINE_GROUP references an AUTOPIPELINE_INCLUDE group, the sum of the pipeline stages inserted into the AUTOPIPELINE_GROUP and the AUTOPIPELINE_INCLUDE group must be between 0 and the AUTOPIPELINE_LIMIT.

When you specify the AUTOPIPELINE_GROUP, AUTOPIPELINE_LIMIT, and AUTOPIPELINE_INCLUDE properties on a register in RTL, the Vivado tools automatically propagate the properties to the net directly connected to the output of the register.

For best timing QoR, Xilinx recommends the following:

- Only apply the AUTOPIPELINE_* properties to registers with no clock enable and no reset control signals.

- Create distinct hierarchies for both sides of the interface, and apply a different USER_SLR_ASSIGNMENT with a different string to each side. The strings must not be `SLR<n>`. The soft floorplanning constraints guide the Vivado placer to move the two groups of registers to different SLRs as needed to improve timing QoR. For example, if hierarchy hierA includes the source registers, and hierB includes the destination registers, you must add the following constraints:

  ```
  set_property USER_SLR_ASSIGNMENT apSrcGrpA [get_cells hierA]
  set_property USER_SLR_ASSIGNMENT apDstGrpB [get_cells hierB]
  ```

IMPORTANT: *The auto-pipelining feature changes the latency of the design. Therefore, you must ensure the functionality remains correct for the specified AUTOPIPELINE_LIMIT range. If the handshake circuitry is required, you must add appropriate logic, such as a FIFO, with enough depth to support backpressure without losing data. The Vivado tools do not verify the correctness of the design logic.*

Note: For the best timing QoR results, the auto-pipeline properties must be set on registers without clock enable or reset logic.

The following figure shows how the auto-pipeline properties are used in the AXI Register Slice RTL.

```
(* autopipeline_group="fwd",autopipeline_limit=24,autopipeline_include="resp" *) reg                    s_handshake_pipe = 1'b0;
                                                                                  reg                    m_handshake_q    = 1'b0;
(* autopipeline_group="resp" *)                                                   reg                    m_ready_pipe     = 1'b0;
                                                                                  reg                    s_ready_i        = 1'b0;
(* autopipeline_group="fwd",autopipeline_limit=24,autopipeline_include="resp" *) reg  [C_DATA_WIDTH-1:0] s_payload_pipe;
                                                                                  reg  [C_DATA_WIDTH-1:0] m_payload_q;
                                                                                  wire                   m_valid_i;
                                                                                  wire                   pop;
```

Figure 2-19: **Example of Auto-Pipelining RTL Property Usage**

The following logic diagram shows one AXI channel of the AXI Register Slice with nets tagged with auto-pipeline properties.

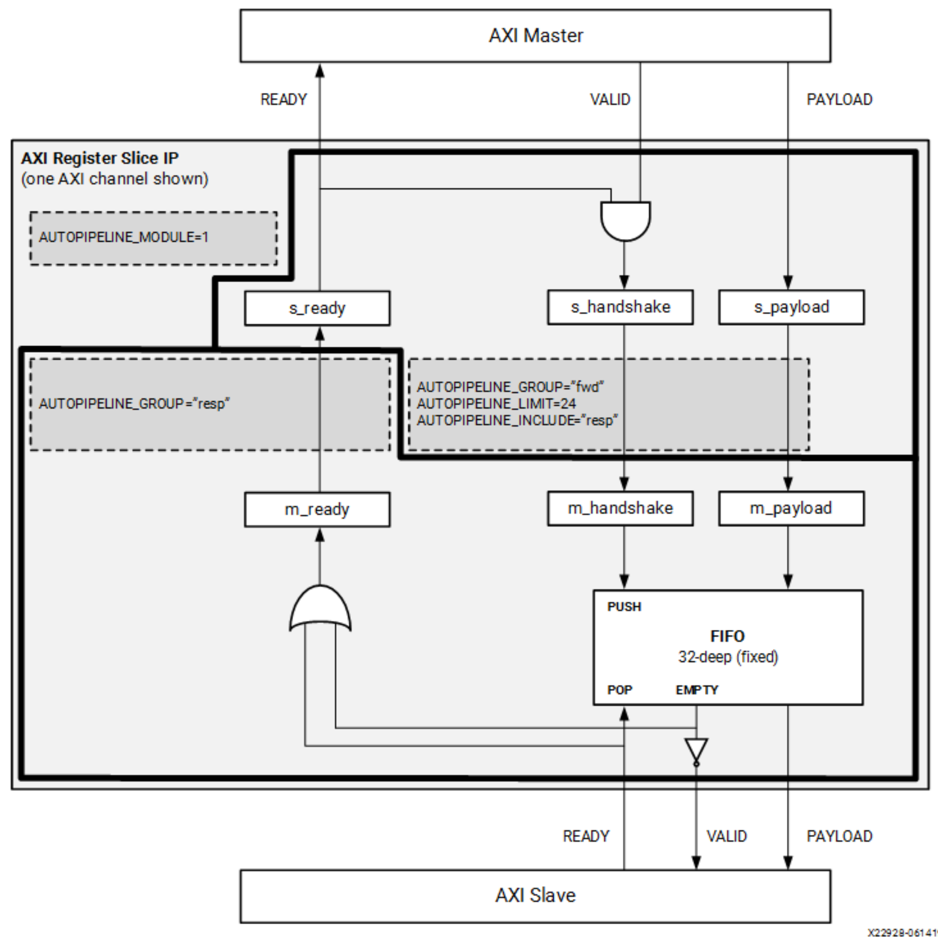


Figure 2-20: **Auto-Pipelining Logic Diagram**

Reviewing the Auto-Pipelining Implementation Results

The following tables are printed in the Vivado log file during the floorplanning phase of `place_design`:

- **Summary of Latency Increase due to Auto-Pipeline Insertion**: This table details the number of pipeline stages inserted for each group.
- **Summary of Physical Synthesis Optimizations**: This table shows the total number of inserted pipeline registers and the number of auto-pipeline groups optimized (Optimized Cells/Nets).

The following figure shows an example of the Summary of Latency Increase Due to Auto-Pipeline Insertion table.

```
Phase 2.1 Floorplanning

Summary of Latency Increase due to Auto-Pipeline Insertion
============================================================
```

Module	Group	Limit	Actual	Include Group
design_1_i/group0/axi_register_slice_0/inst/ar16.ar_auto	fwd	24	9	resp
design_1_i/group0/axi_register_slice_0/inst/ar16.ar_auto	resp	-	9	
design_1_i/group0/axi_register_slice_0/inst/aw16.aw_auto	fwd	24	9	resp
design_1_i/group0/axi_register_slice_0/inst/aw16.aw_auto	resp	-	9	
design_1_i/group0/axi_register_slice_0/inst/b16.b_auto	fwd	24	9	resp
design_1_i/group0/axi_register_slice_0/inst/b16.b_auto	resp	-	9	
design_1_i/group0/axi_register_slice_0/inst/r16.r_auto	fwd	24	9	resp
design_1_i/group0/axi_register_slice_0/inst/r16.r_auto	resp	-	9	
design_1_i/group0/axi_register_slice_0/inst/w16.w_auto	fwd	24	9	resp

Figure 2-21: **Example of Summary of Latency Increase Due to Auto-Pipeline InsertionTable**

The following figure shows an example of the Summary of Physical Synthesis Optimizations table.

```
Summary of Physical Synthesis Optimizations
============================================
```

Optimization	Added Cells	Removed Cells	Optimized Cells/Nets	Dont Touch	Iterations	Elapsed
Auto Pipeline	1582	0	10	68	1	00:00:01
Total	1582	0	10	68	1	00:00:01

Figure 2-22: **Summary of Physical Synthesis Options for Auto Pipeline Table**

The inserted pipeline registers can be retrieved based on their names as follows:

```
<origCellName>_psap and <origCellName>_psap_<N>
```

The following figure shows the path from SLR2 to SLR0 where nine pipeline stages were automatically inserted during `place_design`.

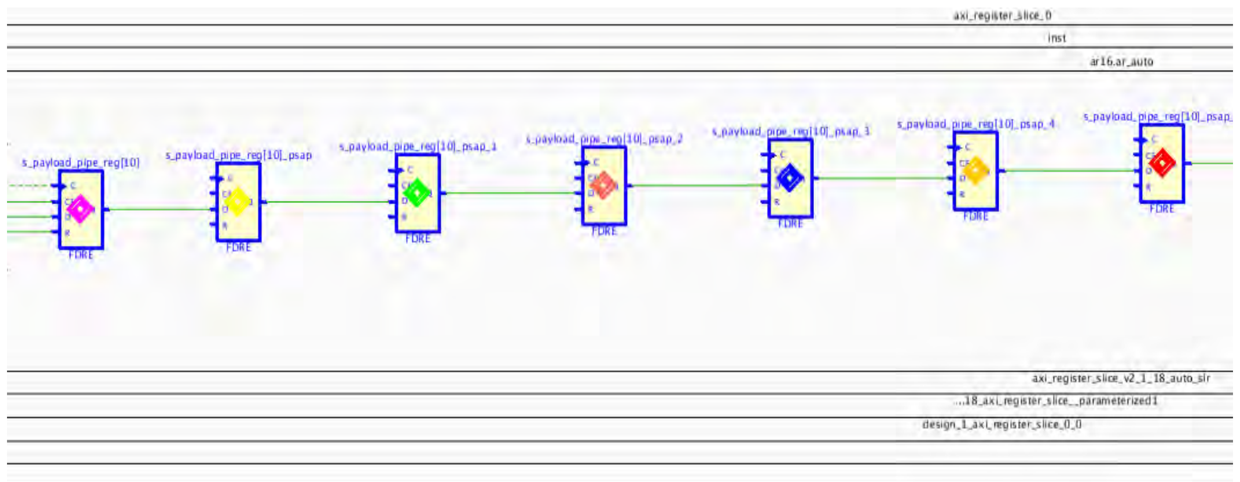


Figure 2-23: **Schematic View of Auto-Pipeline Inserted Registers**

The following figure shows the same example in the Device view.

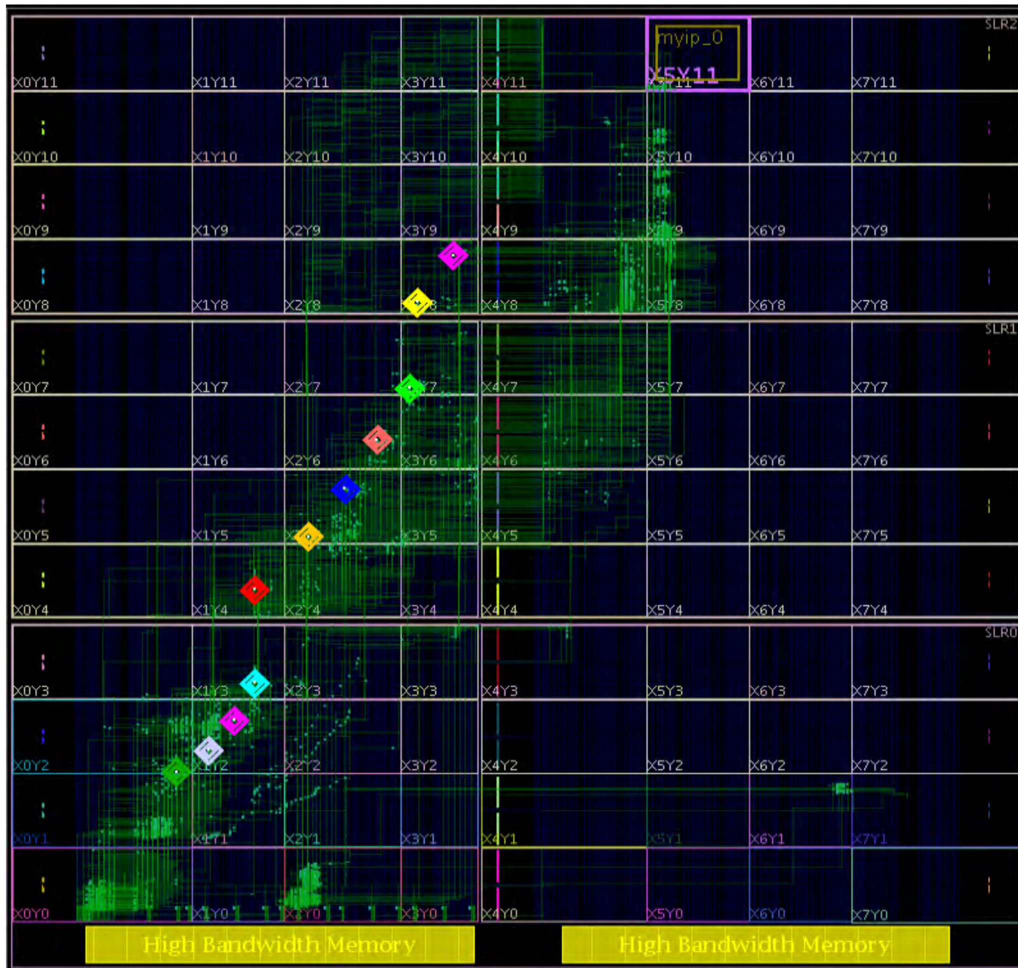


Figure 2-24: **Device View of Auto-Pipeline Inserted Registers**

Physical Optimization

Physical optimization performs timing-driven optimization on the negative-slack paths of a design. Physical optimization has two modes of operation: post-place and post-route.

In post-place mode, optimization occurs based on timing estimates based on cell placement. Physical optimization automatically incorporates netlist changes due to logic optimizations and places cells as needed.

In post-route mode, optimization occurs based on actual routing delays. In addition to automatically updating the netlist on logic changes and placing cells, physical optimization also automatically updates routing as needed.

IMPORTANT: *Post-route physical optimization is most effectively used on designs that have a few failing paths. Using post-route physical optimization on designs with WNS<-0.200 ns or more than 200 failing end points can result in long run time with little improvement to QOR.*

Overall physical optimization is more aggressive in post-place mode, where there is more opportunity for logic optimization. In post-route mode, physical optimization tends to be more conservative to avoid disrupting timing-closed routing. Before running, physical optimization checks the routing status of the design to determine which mode to use, post-place or post-route.

If a design does not have negative slack, and a physical optimization with a timing based optimization option is requested, the command exits quickly without performing optimization. To balance runtime and design performance, physical optimization does not automatically attempt to optimize all failing paths in a design. Only the top few percent of failing paths are considered for optimization. So it is possible to use multiple consecutive runs of physical optimization to gradually reduce the number of failing paths in the design.

Available Physical Optimizations

The Vivado tools perform the physical optimizations on the in-memory design, as shown in the following table.

IMPORTANT: *Physical optimization can be limited to specific optimizations by choosing the corresponding command options. Only those specified optimizations are run, while all others are disabled, even those normally performed by default.*

Table 2-10: **Post-Place and Post-Route Physical Optimizations**

Option Name	post-place		post-route	
	valid	default	valid	default
High-Fanout Optimization	Y	Y	N	n/a
Placement Optimization	Y	Y	Y	Y
Routing Optimization	N	n/a	Y	Y
Rewiring	Y	Y	Y	Y
Critical-Cell Optimization	Y	Y	Y	N
DSP Register Optimization	Y	Y	N	n/a
Block RAM Register Optimization	Y	Y	N	n/a
URAM Register Optimization	Y	N	N	n/a
Shift Register Optimization	Y	Y	N	n/a
Critical Pin Optimization	Y	Y	Y	Y
Block RAM Enable Optimization	Y	Y	N	n/a
Hold-Fixing	Y	N	Y	N
Negative-Edge FF Insertion	Y	N	N	n/a

Table 2-10: **Post-Place and Post-Route Physical Optimizations *(Cont'd)***

Option Name	post-place		post-route	
	valid	default	valid	default
Retiming	Y	N	Y	N
Forced Net Replication	Y	N	N	n/a
SLR-Crossing Optimization	Y	N	Y	Y
Clock Optimization	N	n/a	Y	Y

When an optimization is performed on a primitive cell, the `PHYS_OPT_MODIFIED` property of the cell is updated to reflect the optimizations performed on the cell. When multiple optimizations are performed on the same cell, the `PHYS_OPT_MODIFIED` value contains a list of optimizations in the order they occurred. The following table lists the `phys_opt_design` options that trigger an update to the `PHYS_OPT_MODIFIED` property and the corresponding value.

Table 2-11: **Optimization Options and Values**

phys_opt_design Option	PHYS_OPT_MODIFIED Value
-fanout_opt	FANOUT_OPT
-placement_opt	PLACEMENT_OPT
-slr_crossing_opt	SLR_CROSSING_OPT
-rewire	REWIRE
-insert_negative_edge_ffs	INSERT_NEGEDGE
-critical_cell_opt	CRITICAL_CELL_OPT
-dsp_register_opt	DSP_REGISTER_OPT
-bram_register_opt	BRAM_REGISTER_OPT
-uram_register_opt	URAM_REGISTER_OPT
-shift_register_opt	SHIFT_REGISTER_OPT
-force_replication_on_nets	FORCE_REPLICATION_ON_NETS
-clock_opt	CLOCK_OPT

High-Fanout Optimization

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.

Timing is re-analyzed, and logical changes are committed if timing is improved.

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

Placement-Based Optimization

Optimizes placement on the critical path by re-placing all the cells in the critical path to reduce wire delays.

Routing Optimization

Optimizes routing on critical paths by re-routing nets and pins with shorter delays.

Rewiring

Optimizes 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

Critical-Cell Optimization replicates cells in failing paths. If the loads on a specific cell are placed far apart, the cell might 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

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.

Block RAM Register Optimization

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

URAM Register Optimization

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

Shift Register Optimization

The shift register optimization improves timing on negative slack paths between shift register cells (SRLs) and other logic cells.

If there are timing violations to or from shift register cells (SRL16E or SRLC32E), the optimization extracts a register from the beginning or end of the SRL register chain and places it into the logic fabric to improve timing. The optimization shortens the wirelength of the original critical path.

The optimization only moves registers from a shift register to logic fabric, but never from logic fabric into a shift register, because the latter never improves timing.

The prerequisites for this optimization to occur are:

- The SRL address must be one or greater, such that there are register stages that can be moved out of the SRL.
- The SRL address must be a constant value, driven by logic 1 or logic 0.
- There must be a timing violation ending or beginning from the SRL cell that is among the worst critical paths.

Certain circuit topologies are not optimized:

- SRLC32E that are chained together to form larger shift registers are not optimized.
- SRLC32E using a Q31 output pin.
- SRL16E that are combined into a single LUT with both O5 and O6 output pins used.

Registers moved from SRLs to logic fabric are FDRE cells. The FDRE and SRL INIT properties are adjusted accordingly as is the SRL address. Following is an example.

A critical path begins at a shift register (SRL16E) `srl_inste`, as shown in Figure 2-25.

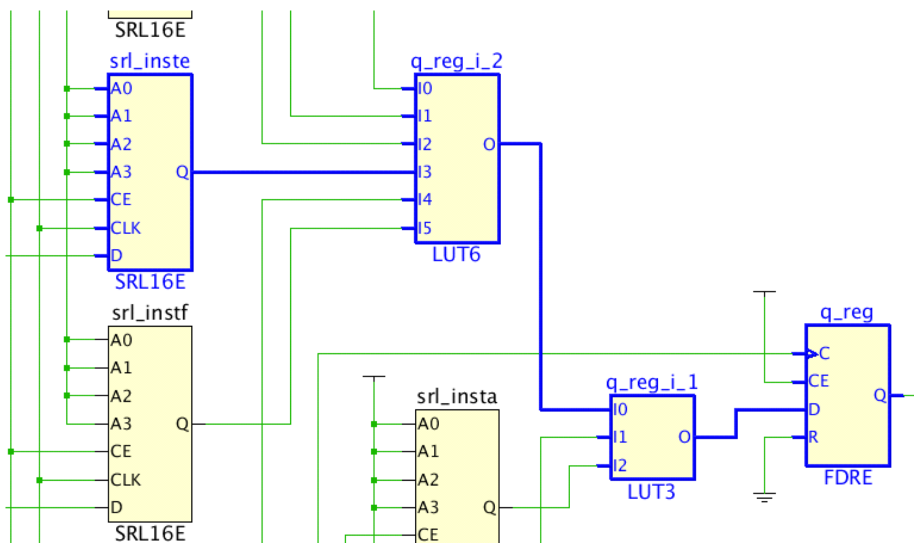


Figure 2-25: **Critical Path Starting at Shift Register srl_inste**

After shift register optimization, the final stage of the shift register is pulled from the SRL16E and placed in the logic fabric to improve timing, as shown in Figure 2-26.

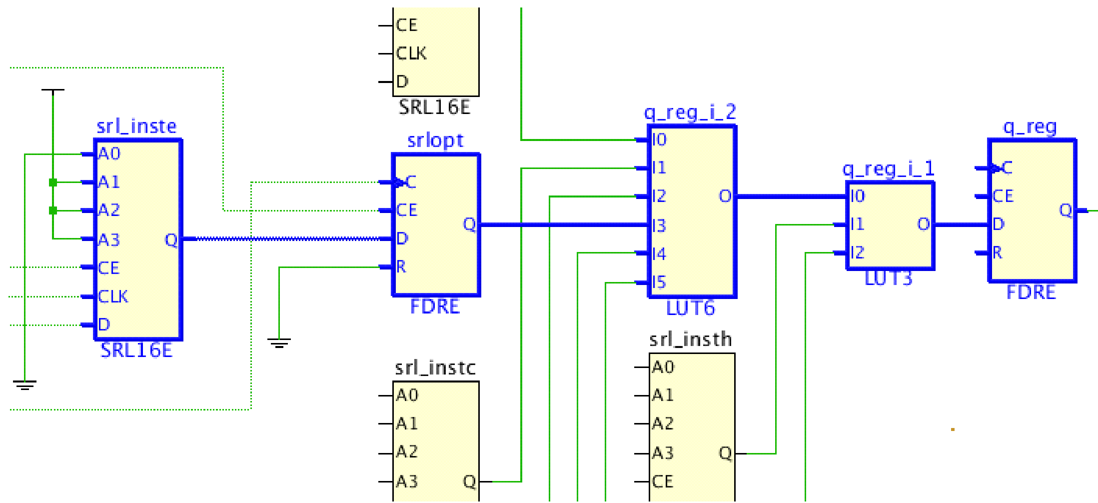


Figure 2-26: **Critical Path after Shift Register Optimization**

The `srl_inste` SRL16E address is decremented to reflect one fewer internal register stage. The original critical path is now shorter as the `srlopt` register is placed closer to the downstream cells and the FDRE cell has a relatively faster clock-to-output delay.

Critical Pin Optimization

Critical Pin Optimization performs remapping of logical LUT input pins to faster physical pins to improve critical path timing. A critical path traversing a logical pin mapped to a slow physical pin such as A1 or A2 is reassigned to a faster physical pin such as A6 or A5 if it improves timing. A cell with a `LOCK_PINS` property is skipped, and the cell retains the mapping specified by `LOCK_PINS`. Logical-to-physical pin mapping is given by the command `get_site_pins`.

Block RAM Enable Optimization

The block RAM enable optimization improves timing on critical paths involving power-optimized block RAMs.

Pre-placement block RAM power optimization restructures the logic driving block RAM read and write enable inputs, to reduce dynamic power consumption. After placement, the restructured logic might become timing-critical. The block RAM enable optimization reverses the enable-logic optimization to improve the slack on the critical enable-logic paths.

Hold-Fixing

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

Aggressive Hold-Fixing

Performs optimizations to insert data path delay to fix hold time violations. This optimization considers significantly more hold violations than the standard hold-fix algorithm.

TIP: *Hold-Fixing only fixes hold time violations above a certain threshold. This is because the router is expected to fix any hold time violations that are less than the threshold.*

Negative-Edge Register Insertion

Inserts negative-edge triggered registers to fix difficult hold time violations. A register insertion splits a hold-critical timing path into two half-period paths, making it easier to meet hold requirements. As the name implies, only negative-edge-triggered register insertion is supported which fixes hold violations between two positive-edge-triggered sequential cells.

Retiming

Retiming improves the delay on the critical path by moving registers across combinational logic. The `phys_opt_design` retiming optimization supports forward retiming.

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.

SLR-Crossing Optimization

Performs post-place or post-route optimizations to improve the path delay of inter-SLR connections. The optimization adjusts the locations of the driver, load, or both along the SLR crossing. Replication is supported in post-route optimization if the driver has inter- and intra-SLR loads. A TNS cleanup option is supported with the `-tns_cleanup` switch in conjunction with the `-slr_crossing_opt` switch. TNS cleanup allows some slack degradation on other paths when performing inter-SLR path optimization as long as the overall WNS of the design does not degrade. For UltraScale devices, either a `TX_REG` or an

RX_REG SLL register can be targeted. In UltraScale+ devices both, TX_REG and RX_REG registers on the same inter-SLR connection can be targeted.

SLL Register Hold Fix

Performs SLL register hold fix optimization for UltraScale+ devices. Use this option when the router is having trouble resolving hold violations on SLR crossing paths between the dedicated SLL TX_REG and RX_REG registers.

Clock Optimization

Inserts global buffers to create useful skew between critical path start and endpoints. To improve setup timing, buffers are inserted to delay the destination clock.

Routing Optimization

Performs routing optimization on timing-critical nets to reduce delay.

Path Group Optimization

Performs post-place and post-route optimizations on the specified path groups only.

TIP: *Use the `group_path` Tcl command to set up the path groups that are targeted for optimization.*

Physical Optimization Messages

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

A summary, as shown in the following figure, is provided at the end of physical optimization showing statistics of each optimization phase and its impact on design performance. This highlights the types of optimizations that are most effective for improving WNS.

Summary of Physical Synthesis Optimizations
===

Optimization	WNS Gain (ns)	TNS Gain (ns)	Added Cells	Removed Cells	Optimized Cells/Nets	Dont Touch	Iterations	Elapsed (s)
Fanout	0.000	0.000	0	0	0	0	3	0
Placement Based	0.000	0.000	0	0	0	0	4	15
Rewire	0.000	0.000	0	0	0	0	3	0
Critical Cell	0.000	0.000	0	0	0	0	3	0
DSP Register	0.000	0.000	0	0	0	0	2	0
BRAM Register	0.000	0.000	0	0	0	0	2	0
Shift Register	0.000	0.000	0	0	0	0	2	0
Very High Fanout	0.000	80.772	37	0	8	0	1	245
Critical Pin	0.000	0.000	0	0	0	0	1	0
Critical Path	0.000	273.445	3	0	119	0	0	260
Total	0.000	354.217	40	0	127	0	22	521

Figure 2-27: **Summary of Physical Synthesis Optimizations**

phys_opt_design

The phys_opt_design command runs physical optimization on the design. It can be run in post-place mode after placement and in post-route mode after the design is fully-routed.

phys_opt_design Syntax

```
phys_opt_design  [-fanout_opt] [-placement_opt] [-routing_opt]
                 [-slr_crossing_opt] [-rewire] [-insert_negative_edge_ffs]
                 [-critical_cell_opt] [-dsp_register_opt] [-bram_register_opt]
                 [-uram_register_opt] [-bram_enable_opt] [-shift_register_opt]
                 [-hold_fix] [-aggressive_hold_fix] [-retime]
                 [-force_replication_on_nets <args>] [-directive <arg>]
                 [-critical_pin_opt] [-clock_opt] [-path_groups <args>]
                 [-tns_cleanup] [-sll_reg_hold_fix] [-quiet] [-verbose]
```

Note: The -tns_cleanup option can only be run in conjunction with the -slr_crossing_opt option.

phys_opt_design Example Script

```
open_checkpoint top_placed.dcp

# Run post-place phys_opt_design and save results
phys_opt_design
write_checkpoint -force $outputDir/top_placed_phys_opt.dcp
report_timing_summary -file $outputDir/top_placed_phys_opt_timing.rpt

# Route the design and save results
route_design
write_checkpoint -force $outputDir/top_routed.dcp
report_timing_summary -file $outputDir/top_routed_timing.rpt

# Run post-route phys_opt_design and save results
phys_opt_design
write_checkpoint -force $outputDir/top_routed_phys_opt.dcp
report_timing_summary -file $outputDir/top_routed_phys_opt_timing.rpt
```

The phys_opt_design example script runs both post-place and post-route physical optimization. First, the placed design is loaded from a checkpoint, followed by post-place phys_opt_design. The checkpoint and timing results are saved. Next the design is routed, with progress saved afterwards. That is followed by post-route phys_opt_design and saving the results. Note that the same command phys_opt_design is used for both post-place and post-route physical optimization. No explicit options are used to specify the mode.

Using Directives

Directives provide different modes of behavior for the phys_opt_design command. Only one directive can be specified at a time, and the directive option is incompatible with other options. The available directives are described below.

- `Explore`:
  Run different algorithms in multiple passes of optimization, including replication for very high fanout nets, SLR crossing optimization, and a final phase called Critical Path Optimization where a subset of physical optimizations are run on the top critical paths of all endpoint clocks, regardless of slack.
- `ExploreWithHoldFix`:
  Run different algorithms in multiple passes of optimization, including hold violation fixing, SLR crossing optimization and replication for very high fanout nets.
- `ExploreWithAggressiveHoldFix`:
  Run different algorithms in multiple passes of optimization, including aggressive hold violation fixing, SLR crossing optimization and replication for very high fanout nets.

TIP: *Hold-Fixing only fixes hold time violations above a certain threshold. This is because the router is expected to fix any hold time violations that are less than the threshold.*

- `AggressiveExplore`:
  Similar to `Explore` but with different optimization algorithms and more aggressive goals. Includes a SLR crossing optimization phase that is allowed to degrade WNS which should be regained in subsequent optimization algorithms. Also includes a hold violation fixing optimization.
- `AlternateReplication`:
  Use different algorithms for performing critical cell replication.
- `AggressiveFanoutOpt`:
  Uses different algorithms for fanout-related optimizations with more aggressive goals.
- `AddRetime`:
  Performs the default `phys_opt_design` flow and adds register retiming.
- `AlternateFlowWithRetiming`:
  Perform more aggressive replication and DSP and block RAM optimization, and enable register retiming.
- `Default`:
  Run `phys_opt_design` with default settings.
- `RuntimeOptimized`:
  Run fewest iterations, trade higher design performance for faster run time.

TIP: *All directives are compatible with both post-place and post-route versions of* `phys_opt_design`.

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` property during physical optimization. It does not perform physical optimization on nets or cells with these properties. To speed up the net selection process, nets with `DONT_TOUCH` properties are pre-filtered and not considered for physical optimization. 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 this link in the *Vivado Design Suite User Guide: Synthesis* (UG901) [Ref 8].

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 can still occur within the cell.

The tools automatically add `DONT_TOUCH` properties of value `TRUE` to nets that have `MARK_DEBUG` properties of value `TRUE`. This is done to keep the nets intact throughout the implementation flow so that they can be probed at any design stage. This is the recommended use of `MARK_DEBUG`. However there might be rare occasions on which the `DONT_TOUCH` is too restrictive and prevents optimizations such as replication and retiming, leading to more difficult timing closure. In those cases `DONT_TOUCH` can be set to a value of `FALSE` while keeping `MARK_DEBUG TRUE`. The consequence of removing the `DONT_TOUCH` properties is that nets with `MARK_DEBUG` can be optimized away and no longer probed. If a `MARK_DEBUG` net is replicated, only the original net retains `MARK_DEBUG`, not the replicated nets.

Physical Optimization Reports

The Tcl reporting command `report_phys_opt` provides details of each optimization performed by `phys_opt_design` at a very fine level of detail. It must be run in the same Vivado session as `phys_opt_design` while the optimization history resides in memory. Therefore, if a report is desired, it is recommended to include the `report_phys_opt` command in Tcl scripts immediately following the last `phys_opt_design` command.

The reports are available only for post-placement `phys_opt_design` optimizations. The reports are cumulative, reflecting all `phys_opt_design` optimizations, including those from multiple runs of `phys_opt_design`.

The following report example shows the first entry of a fanout optimization involving a register named `pipeline_en`. The following details are shown in the report:

1. The original driver `pipeline_en` drives 816 loads and the paths containing this high fanout net fail timing with WNS of -1.057 ns.
2. The driver `pipeline_en` was replicated to create one new cell, `pipeline_en_replica`.
3. The 816 loads were split between `pipeline_en_replica`, which takes 386 loads, and the original driver `pipeline_en`, which takes the remaining 430 loads.
4. After replication and placement of `pipeline_en_replica`, the WNS of `pipeline_en_replica` paths is +0.464 ns, and the WNS of `pipeline_en` paths is reduced to zero.
5. The placement of the original driver `pipeline_en` was changed to improve WNS based on the locations of its reduced set of loads.

```
Fanout Optimization
============================

-----------------------------------------
|    Number of Nets Optimized :   |  8  |
|  Number of New Cells Created :  |  37 |
-----------------------------------------

-----------------------------------------------------------------------------------------------------------------------
| Original Cell      | New/Modified Cell             | #Loads |  WNS    | TNS      | Location                     |
-----------------------------------------------------------------------------------------------------------------------
| pipeline_en        |                             - |   816  | -1.057  | 2156.54  | SLICE_X404Y250 SLICEM.CFF    |
|                  - | pipeline_en_replica           |   386  |  0.464  |  1238.7  | SLICE_X364Y262 SLICEM.AFF    |
|                  - | pipeline_en                   |   430  |      0  |       0  | SLICE_X419Y253 SLICEL.AFF    |
```

Figure 2-28: **Fanout Optimization Report**

Interactive Physical Optimization

Beginning with the 2015.3 release, Physical Optimization has the capability to "replay" optimization using an interactive Tcl command `iphys_opt_design`. The `iphys_opt_design` command describes a specific optimization occurrence, such as a the replication of a critical cell or the pulling of a set of registers from a block RAM. The command includes all the information necessary to recreate both the netlist and the placement changes required for the optimization occurrence.

Interactive physical optimization can be used in two ways:

- Applying post-placement physical optimizations to the pre-placement netlist to improve the overall placement result and improve design performance.
- Saving the physical optimizations in a Tcl script to be repeated as needed

Retrofitting phys_opt_design Netlist Changes

The design flow involving retrofit is described in the following figure.

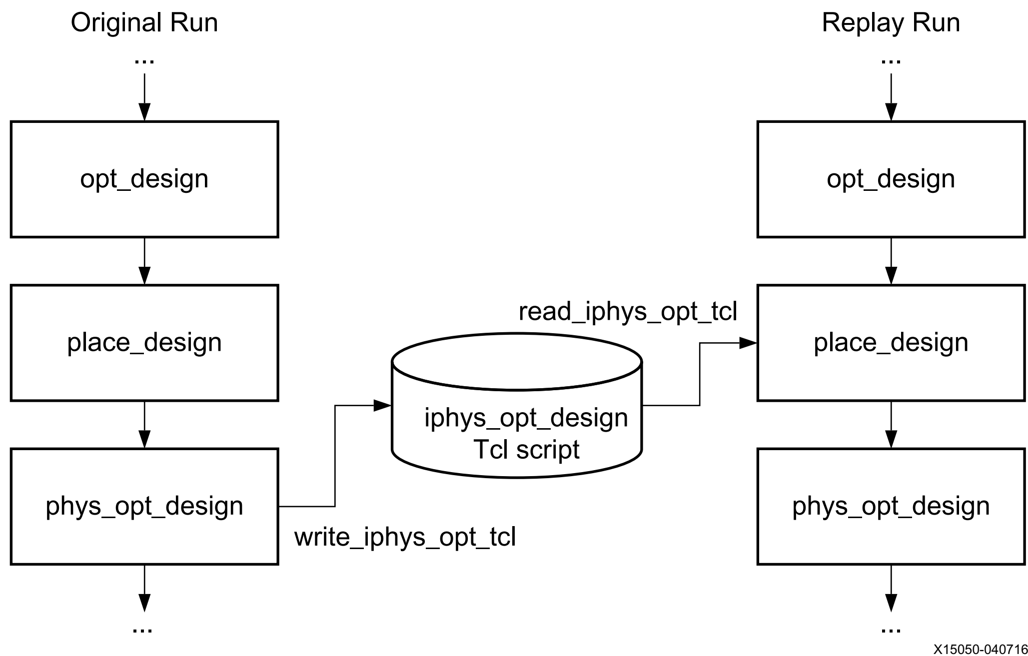


Figure 2-29: **Design Flow Involving Retrofit**

Two runs are involved, which are the “original run,” where `phys_opt_design` is run after `place_design` and the “replay run,” where `phys_opt_design` netlist changes are performed before placement.

After the original run, the `phys_opt_design` optimizations are saved to a Tcl script file using the Tcl command `write_iphys_opt_tcl`. The script contains a series of `iphys_opt_design` Tcl commands to recreate exactly the design changes performed by `phys_opt_design` in the original run. You can save the optimizations from the current design in memory or after opening an implemented design or checkpoint where `phys_opt_design` has performed optimization.

The same design and constraints are used for the replay run. Before `place_design` runs, the `read_iphys_opt_tcl` command processes the `iphys_opt_design` command script and applies the netlist changes from the original run. As a result of the netlist changes, the design in the replay run might be more suitable for placement than the original run. The design now incorporates the benefits of the `phys_opt_design` optimizations before placement, such as fewer high-fanout nets after replication and fewer long distance paths from block RAM outputs.

Similar to the `phys_opt_design` command, the `read_iphys_opt_tcl` command has options to limit the replayed design steps to certain types, such as fanout optimization, block RAM register optimization, and rewiring.

Repeating phys_opt_design Design Changes

The design flow for repeating `phys_opt_design` design changes is shown in the following figure.

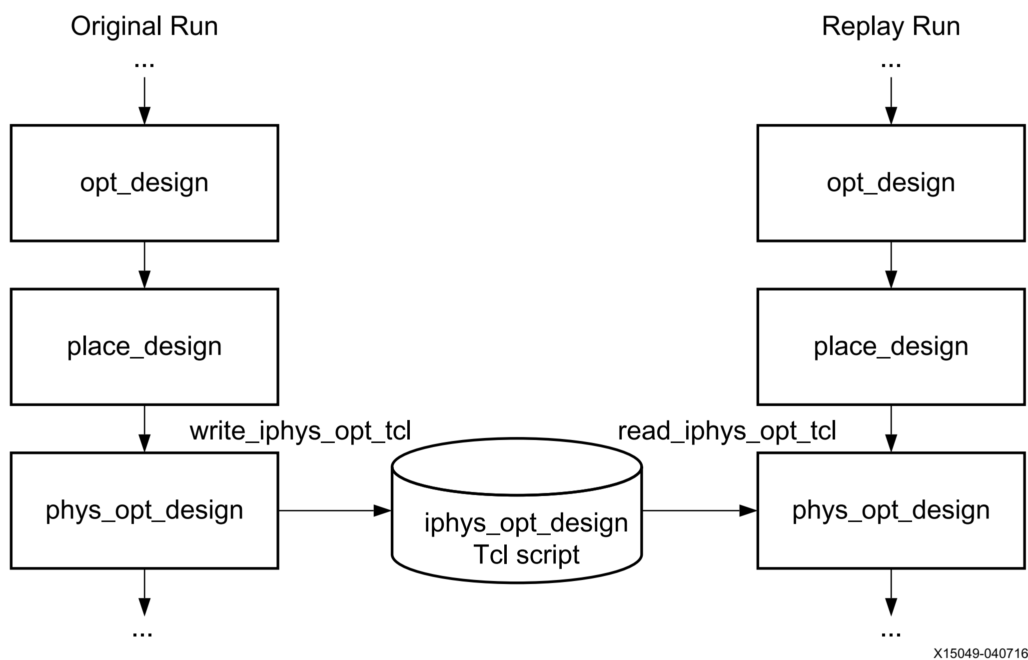


Figure 2-30: **Design Flow when Repeating phys_opt_design Changes**

This flow differs from the retrofit flow in two aspects:

- The `iphys_opt_design` changes are incorporated after `place_design` instead of beforehand.
- Both placement changes as well as netlist changes are captured in the `iphys_opt_design` Tcl script.

Typically, you would use this flow to gain more control over the post-place `phys_opt_design` step. Custom "recipes" are created from combinations of replayed optimizations and new optimizations resulting in many possibilities for exploration of design closure.

The `write_iphys_opt_tcl` and `read_iphys_opt_tcl` commands have a `-place` option to replay the placement changes from `phys_opt_design`. This option should be used in this flow to repeat `phys_opt_design` steps after placement.

Interactive Physical Optimization Command Reference

The interactive physical optimization commands, along with corresponding options, are described below.

write_iphys_opt_tcl

This command writes a file containing the `iphys_opt_design` Tcl commands corresponding to the physical optimizations performed in the current design.

Syntax:

```
write_iphys_opt_tcl [-place] [-quiet] [-verbose] <output file>
```

The `-place` option directs the command to include placement information with the `iphys_opt_tcl` commands. Use this option when you intend to apply placement with netlist changes during `iphys_opt_design` command replay.

The `write_iphys_opt_tcl` command can be used any time after `phys_opt_design` has been run.

read_iphys_opt_tcl

This command reads a file containing the `iphys_opt_design` Tcl commands corresponding to the physical optimizations performed in a previous run.

Syntax:

```
read_iphys_opt_tcl  [-fanout_opt] [-critical_cell_opt] [-placement_opt]
                    [-rewire] [-dsp_register_opt] [-bram_register_opt]
                    [-uram_register_opt][-shift_register_opt] [-critical_pin_opt]
                    [-insert_negative_edge_ffs] [-slr_crossing_opt]
                    [-include_skipped_optimizations] [-quiet]
                    [-verbose] <input file>
```

The `read_iphys_opt_design` command has many of the same options as `phys_opt_design` to limit the scope of replayed optimizations to only those specified. These options include `-fanout_opt`, `-critical_cell_opt`, `-placement_opt`, `-rewire`, `-dsp_register_opt`, `-bram_register_opt`, `-uram_register_opt`, `-shift_register_opt`, `-insert_negative_edge_ffs`, `-slr_crossing_opt`, and `-critical_pin_opt`.

Apply the skipped optimizations that are defined in the input Tcl script, as well as the standard optimizations. These are optimizations identified by `phys_opt_design` that are skipped because suitable locations for optimized logic cannot be found. When this option is specified, the `iphys_opt_design` command will attempt to use the included skipped optimizations in the pre-placement netlist.

iphys_opt_design

The `iphys_opt_design` command is a low-level Tcl command that performs a physical optimization. All default `phys_opt_design` optimizations can be performed using `iphys_opt_design`. Although it is possible to modify `iphys_opt_design` commands, and even to create them from scratch, you would typically write them to a script and replay them in a separate run.

RECOMMENDED: *Avoid using the Tcl source command to execute a script of* `iphys_opt_design` *commands. For most efficient processing of commands and for fastest runtime, use the* `read_iphys_opt_tcl` *command instead.*

Syntax:

```
iphys_opt_design  [-fanout_opt] [-critical_cell_opt] [-placement_opt] [-rewire]
                  [-net <arg>] -cluster <args> -place_cell <args> [-place]
                  [-dsp_register_opt] [-bram_register_opt] [-uram_register_opt]
                  [-shift_register_opt] [-slr_crossing_opt] [-cell <arg>]
                  [-packing][-unpacking][-port <arg>] [-critical_pin_opt]
                  [-skipped_optimization][-insert_negative_edge_ffs]
                  [-quiet] [-verbose]
```

Routing

The Vivado router performs routing on the placed design, and performs optimization on the routed design to resolve hold time violations.

The Vivado router starts with a placed design and attempts to route all nets. It can start with a placed design that is unrouted, partially routed, or fully routed.

For a partially routed design, the Vivado router uses the existing routes as the starting point, instead of starting from scratch. For a fully-routed design, the router checks for timing violations and attempts to re-route critical portions to meet timing.

Note: The re-routing process is commonly referred to as "rip-up and re-route."

The router provides options to route the entire design or to route individual nets and pins.

When routing the entire design, the flow is timing-driven, using automatic timing budgeting based on the timing constraints.

Routing individual nets and pins can be performed using two distinct modes:

- Interactive Router mode
- Auto-Delay mode

The Interactive Router mode uses fast, lightweight timing modeling for greater responsiveness in an interactive session. Some delay accuracy is sacrificed with the estimated delays being pessimistic. Timing constraints are ignored in this mode, but there are several choices to influence the routing:

- Resource-based routing (default): The router chooses from the available routing resources, resulting in the fastest router runtime.

- Smallest delay (the `-delay` option): The router tries to achieve the smallest possible delay from the available routing resources.
- Delay-driven (the `-max_delay` and `-min_delay` options): Specify timing requirements based on a maximum delay, minimum delay, or both. The router tries to route the net with a delay that meets the specified requirements.

In Auto-Delay mode, the router runs the timing-driven flow with automatic timing budgeting based on the timing constraints, but unlike the default flow, only the specified nets or pins are routed. This mode is used to route critical nets and pins before routing the remainder of the design. This includes nets and pins that are setup-critical, hold-critical, or both. Auto-Delay mode is not intended for routing individual nets in a design containing a significant amount of routing. Interactive routing should be used instead.

For best results when routing many individual nets and pins, prioritize and route these individually. This avoids contention for critical routing resources.

Routing requires a one-time “run time hit” for initialization, even when editing routes of nets and pins. The initialization time increases with the size of the design and with the size of the device. The router does not need to be re-initialized unless the design is closed and reopened.

Design Rule Checks

Before starting routing, the Vivado tools run 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, and 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

Post-routing timing violations are sometimes 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.

Common examples of the impact of poor timing constraints include:

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

RECOMMENDED: *Review timing constraints and correct those that are invalid (or consider RTL changes) before exploring multiple routing options. For more information, see this link in UltraFast Design Methodology Guide for the Vivado Design Suite (UG949) [Ref 13].*

Router Timing Summary

At the end of the routing process, the router reports an estimated timing summary calculated using actual routing delays. However, to improve run time, the router uses incremental timing updates rather than doing the full timing computation to calculate the timing summary. Consequently, the estimated WNS can be more pessimistic (by a few ps) than actual timing. It is therefore possible for the router WNS to be negative while the actual WNS is positive. If the router reports estimated WNS that is negative, the message is a warning, not a critical warning.

TIP: *When you run `route_design -directive Explore`, the router timing summary is based on signoff timing.*

IMPORTANT: *You must check the actual signoff timing using `report_timing_summary` or run `route_design` with the `-timing_summary` option.*

route_design

The `route_design` command runs routing on the design.

route_design Syntax

```
route_design  [-unroute] [-release_memory] [-nets <args>] [-physical_nets]
              [-pins <arg>] [-directive <arg>] [-tns_cleanup]
              [-no_timing_driven] [-preserve] [-delay] [-auto_delay]
              [-max_delay <arg>] [-min_delay <arg>] [-timing_summary] [-finalize]
              [-ultrathreads] [-quiet] [-verbose]
```

Using Directives

When routing the entire design, directives provide different modes of behavior for the `route_design` command. Only one directive can be specified at a time. The directive option is incompatible with most other options to prevent conflicting optimizations. The following directives are available:

- `Explore`:
  Allows the router to explore different critical path placements after an initial route.
- `AggressiveExplore`:
  Directs the router to further expand its exploration of critical path routes while maintaining original timing budgets. The router runtime might be significantly higher compared to the Explore directive because the router uses more aggressive optimization thresholds to attempt to meet timing constraints.
- `NoTimingRelaxation`:
  Prevents the router from relaxing timing to complete routing. If the router has difficulty meeting timing, it runs longer to try to meet the original timing constraints.
- `MoreGlobalIterations`:
  Uses detailed timing analysis throughout all stages instead of just the final stages, and runs more global iterations even when timing improves only slightly.
- `HigherDelayCost`:
  Adjusts the internal cost functions of the router to emphasize delay over iterations, allowing a tradeoff of run time for better performance.
- `RuntimeOptimized`:
  Run fewest iterations, trade higher design performance for faster run time.
- `AlternateCLBRouting`:
  Chooses alternate routing algorithms that require extra runtime but may help resolve routing congestion.
- `Quick`:
  Absolute, fastest compile time, non-timing-driven, performs the minimum required for a legal design.
- `Default`:
  Run `route_design` with default settings.

Trading Compile Time for Better Routing

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

- `NoTimingRelaxation`
- `MoreGlobalIterations`
- `HigherDelayCost`
- `AdvancedSkewModeling`
- `AggressiveExplore`

Using Other route_design Options

Following are more details on the `route_design` options and option values where applicable.

- Using `-nets`

  This limits operation to only the list of nets specified. The option requires an argument that is a Tcl list of net objects. Note that the argument must be a net object, the value returned by `get_nets`, as opposed to the string value of the net names.

- Using `-pins`

  This limits operation only to the specified pins. The option requires an argument, which is a Tcl list of pin objects. Note that the argument must be a pin object, the value returned by `get_pins`, as opposed to the string value of the pin names.

- Using `-delay`

  By default, the router routes individual nets and pins with the fastest run time, using available resources without regard to timing criticality. The -delay option directs the router to find the route with the smallest possible delay.

- Using `-min_delay` and `-max_delay`

  These options can be used only with the pin option and to specify a desired target delay in picoseconds. The `-max_delay` option specifies the maximum desired slow-max corner delay for the routing of the specified pin. Similarly the `-min_delay` option specifies the minimum fast-min corner delay. The two options can be specified simultaneously to create a desired delay range.

- Using `-auto_delay`

  Use with `-nets` or `-pins` option to route in timing constraint-driven mode. Timing budgets are automatically derived from the timing constraints so this option is not compatible with `-min_delay`, `-max_delay`, or -delay.

- Using `-preserve`

  This option routes the entire design while preserving existing routing. Without `-preserve`, the existing routing is subject to being unrouted and re-routed to improve critical-path timing. This option is most commonly used when "pre-routing" critical nets, that is, routing certain nets first to ensure that they have best access to routing resources. After achieving those routes, the `-preserve` option ensures they are not disrupted while routing the remainder of the design. Note that `-preserve` is completely independent of the `FIXED_ROUTE` and `IS_ROUTE_FIXED` net properties. The route preservation lasts only for the duration of the `route_design` operation in which is it used. The `-preserve` option can be used with `-directive`, with one

exception, the `-directive Explore` option, which modifies placement, which in turn modifies routing.

- Using `-unroute`

  The `-unroute` option removes routing for the entire design or for nets and pins, when combined with the nets or pin options. The option does not remove routing for nets with `FIXED_ROUTE` properties. Removing routing on nets with `FIXED_ROUTE` properties requires the properties to be removed first.

- Using `-timing_summary`

  The router outputs a final timing summary to the log, based on its internal estimated timing which might differ slightly from the actual routed timing due to pessimism in the delay estimates. The `-timing_summary` option forces the router to call the Vivado static timing analyzer to report the timing summary based on the actual routed delays. This incurs additional run time for the static timing analysis. The `-timing_summary` is ignored when the `-directive Explore` option is used.

  When the `-directive Explore` option is used, routing always calls the Vivado static timing analyzer for the most accurate timing updates, whether or not the `-timing_summary` option is used.

- Using `-tns_cleanup`

  For optimal run time, the router focuses on improving the Worst Negative Slack (WNS) path as opposed to reducing the Total Negative Slack (TNS). The `-tns_cleanup` option invokes an optional phase at the end of routing, during which the router attempts to fix all failing paths to reduce the TNS. Consequently, this option might reduce TNS at the expense of run time but might not affect WNS. Use the `-tns_cleanup` option during routing when you intend to follow router runs with post-route physical optimization. Use of this option during routing ensures that physical optimization focuses on the WNS path and that effort is not wasted on non-critical paths that can be fixed by the router. Running `route_design -tns_cleanup` on an already routed design only invokes the TNS cleanup phase of the router and does not affect WNS (TNS cleanup is re-entrant). This option is compatible with `-directive`.

- Using `-physical_nets`

  The `-physical_nets` option operates only on logic 0 and logic 1 routes. The option covers all logic constant values in the design and is compatible with the `-unroute` option. Because constant 0 and 1 tie-offs in the physical device have no exact correlation to logical nets, these nets cannot be routed and unrouted reliably using the `-nets` and `-pins` options.

- Using `-ultrathreads`

This option shortens router runtime at the expense of repeatability. With `-ultrathreads`, the router runs faster but there is a very small variation in routing between identical runs.

- Using `-release_memory`

  After router initialization, router data is kept in memory to ensure optimal performance. This option forces the router to delete its data from memory and release the memory back to the operating system. This option should not be required for mainstream use and is provided in case router memory must be manually managed, for example, with extremely large designs.

- Using `-finalize`

  When routing interactively you can specify `route_design -finalize` to complete any partially routed connections.

  For UltraScale+ designs, this step is required if placement and routing of registers was changed as part of an ECO task.

- Using `-no_timing_driven`

  This option disables timing-driven routing and is used primarily for testing the routing feasibility of a design.

- Using `-eco`

  This option is used with incremental mode to get a shorter runtime after some ECO modifications to the design while keeping the routability and timing closure.

Routing Example Script 1

```
# 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 performs the following steps:

1. Routes the design
2. Writes a design checkpoint after completing routing
3. Generates a timing summary report
4. Writes the report to the specified file.

Routing is performed as part of an implementation run, or by running `route_design` after `place_design` as part of a Tcl script.

The router provides info in the log to indicate progress, such as the current phase (initialization, global routing iterations, and timing updates). At the end of global routing, the log includes periodic updates showing the current number of overlapping nets as the router attempts to achieve a fully legalized design. For example:

```
Phase 4.1 Global Iteration 0
 Number of Nodes with overlaps = 435
 Number of Nodes with overlaps = 3
 Number of Nodes with overlaps = 1
 Number of Nodes with overlaps = 0
```

The timing updates are provided throughout the flow showing timing closure progress.

Timing Summary

```
[Route 35-57] Estimated Timing Summary | WNS=0.105  | TNS=0 | WHS=0.051 | THS=0
```

where:

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

Note: Hold time analysis can be skipped during intermediate routing phases. If hold time is not performed, the router shows a value of "N/A" for WHS and THS.

After routing is complete, the router reports a routing utilization summary and a final estimated timing summary. An example of the Router Utilization Summary is shown below.

Router Utilization Summary

```
Global Vertical Routing Utilization    = 15.3424 %
Global Horizontal Routing Utilization  = 16.3981 %
Routable Net Status*
*Does not include unroutable nets such as driverless and loadless.
Run report_route_status for detailed report.
Number of Failed Nets               = 0
Number of Unrouted Nets             = 0
Number of Partially Routed Nets     = 0
Number of Node Overlaps             = 0
```

Routing Example Script 2

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

# route $preRoutes first with the smallest possible delay
route_design -nets [get_nets $preRoutes] -delay

# preserve the routing for $preRoutes and continue with the rest of the design
route_design -preserve
```

In this example script, a few critical nets are routed first, followed by routing of the entire design. It illustrates routing individual nets and pins (nets in this case), which is typically done 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 `route_design` command initializes the router and routes essential nets, such as clocks.

Routing Example Script 3

```
# get nets of the top 10 setup-critical paths
set preRoutes [get_nets -of [get_timing_paths -max_paths 10]]

# get nets of the top 10 hold-critical paths
lappend preRoutes [get_nets -of [get_timing_paths -hold -max_paths 10]]

# route $preRoutes based on timing constraints
route_design -nets [get_nets $preRoutes] -auto_delay

# preserve the routing for $preRoutes and continue with the rest of the design
route_design -preserve
```

As in example 2, a few critical nets are routed first, followed by routing of the entire design. The difference is the use of `-auto_delay` instead of `-delay`. The router performs timing-driven routing of the critical nets, which sacrifices some runtime for greater accuracy. This is particularly useful for situations in which nets are involved in both setup-critical and hold-critical paths, and the routes must fall within a delay range to meet both setup and hold requirements.

Routing Example Script 4

```
route_design
# 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 -preserve
```

The strategy in this example script illustrates one possible way to address timing failures due to congestion. In the example design, some critical nets represented by `$myCritNets` need routing resources in the same device region as the nets in instance `u0/u1`. The nets in `u0/u1` are not as timing-critical, so they are unrouted to allow the critical nets `$myCritNets` to be routed first, with the smallest possible delay. Then `route_design -preserve` routes the entire design. The `-preserve` switch preserves the routing of `$myCritNets` while the unrouted `u0/u1` nets are re-routed. Table 2-12 summarizes the commands in the example.

Router Messaging

The router provides helpful messages when it struggles to meet timing goals due to congestion or excessive hold violation fixing. The router commonly exhibits these symptoms when it struggles:

- Excessive runtimes, on the order of hours per iteration
- Large number of overlaps reported, in the hundreds or thousands
- Setup and hold slacks become progressively worse, as seen in the Estimated Timing Summaries

The router might provide further warning messages when any of the following occurs:

- Congestion is expected to have negative timing closure impact, which typically occurs when the congestion level is 5 or greater. Level 5 indicates a congested region measuring 32x32 (2^5 = 32).
- The overall router hold-fix effort is expected to be very high, which impacts the ability to meet overall setup requirements.
- Specific endpoint pins become both setup-critical and hold-critical and it is difficult or impossible to satisfy both. The message includes the names of up to ten pins for design analysis.
- Specific CLBs experience high pin utilization or high routing resource utilization which results in local congestion. The messages will include the names of up to ten of the most congested CLBs.
- In extreme cases with severe congestion, the router warns that congestion is preventing the router from routing all nets, and the router will prioritize the successful completion of routing all nets over timing optimizations.

When targeting UltraScale devices or later, the router generates a table showing initial estimated congestion when congestion might affect timing closure. The table does not show specific regions but gives a measure of different types of congestion for an overall assessment. The congestion is categorized into bins of Global (design-wide), Long (connections spanning several CLBs), and Short Congestion. The tables of different runs can be compared to determine which have better chances of meeting performance goals without being too negatively impacted by congestion.

```
INFO: [Route 35-449] Initial Estimated Congestion
 _____________________________________________________________________
|           | Global Congestion | Long Congestion   | Short Congestion  |
|           |___________________|___________________|___________________|
| Direction | Size    | % Tiles | Size    | % Tiles | Size    | % Tiles |
|___________|_________|_________|_________|_________|_________|_________|
|      NORTH|      8x8|     0.13|      4x4|     0.11|    32x32|     1.14|
|___________|_________|_________|_________|_________|_________|_________|
|      SOUTH|      8x8|     0.21|      4x4|     0.35|    16x16|     1.03|
|___________|_________|_________|_________|_________|_________|_________|
```

```
|        EAST|      2x2|        0.05|      2x2|        0.15|      8x8|        0.97|
|____________|_________|____________|_________|____________|_________|____________|
|        WEST|      2x2|        0.03|      2x2|        0.17|      8x8|        0.83|
|____________|_________|____________|_________|____________|_________|____________|
```

Report Design Analysis provides complexity and congestion analysis that can give further insight into the causes of congestion and potential solutions. The congestion reporting also includes an Average Initial Routing Congestion, which is not exactly the same as the congestion reported by the router, but can be analyzed against the pre-route design to determine which regions are causing problems. For further information on Report Design Analysis, refer to the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 10].

Intermediate Route Results

Even when routing fails, the router continues and tries to provide a design that is as complete as possible to aid in debug. If the routing is not complete, you might have to intervene manually.

Use the `report_route_status` command to identify nets with routing errors. For more information see this link in the *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949) [Ref 13].

The router reports routing congestion during Route finalize. The highest congested regions are listed for each direction (North, East, South, and West). For each region, the information includes the dimensions in routing tiles, the routing utilization labeled "Max Cong," and the bounding box coordinates (lower-left corner to upper-right corner). The “INT_xxx” numbers are the coordinates of the interconnecting routing tiles that are visible in the device routing resource view.

Table 2-12: **Commands Used During Routing for Design Analysis**

Command	Function
report_route_status	Reports route status for nets
report_timing	Performs path endpoint analysis
report_design_analysis	Provides information about congested areas

For a complete description of the Tcl reporting commands and their options, see the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 19].

Incremental Implementation

Incremental Implementation refers to the implementation phase of the Incremental Compile design flow that:

- Preserves QoR predictability by reusing prior placement and routing from a reference design.
- Speeds up place and route run time or attempts last mile timing closure.

A diagram of the Incremental Implementation design flow is provided in the following figure.

Note: This diagram also illustrate the incremental synthesis flow. For more details about incremental synthesis flow, see the "Incremental Synthesis" section in the *Vivado Design Suite User Guide: Synthesis* (UG901)[Ref 8].

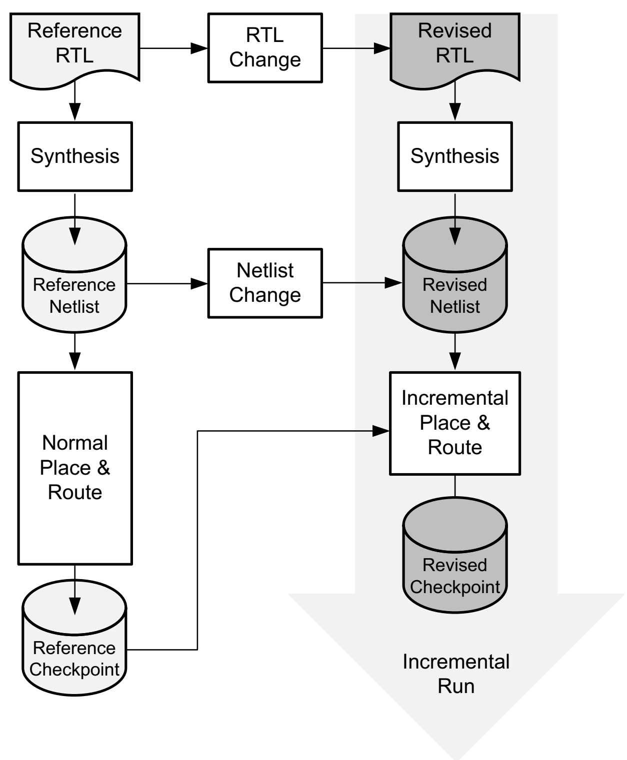


Figure 2-31: **Incremental Compile Design Flow**

Incremental Implementation Flow Designs

As shown in Figure 2-31, the Incremental Run requires a reference checkpoint to be read in to start the incremental place and route. The `read_checkpoint -incremental <reference>.dcp` command initiates the incremental flow and must be issued before place_design.

Reference Design

The reference design is typically a fully routed checkpoint from a previous iteration or a different variation of the incremental design. If using a different variation or a design, it is important that the hierarchy names from the reference design match the incremental design.

When lower levels of reuse are required, for example reusing only RAM and DSP block placement, it is acceptable to have as little as the placement information for those cells in the reference checkpoint. The reference design must match device. It is also recommended to match tool version but this is not a requirement.

Incremental Design

The incremental design is the updated design that is to be run through the implementation tools. It can include RTL Changes, Netlist Changes or both.

Constraint changes are allowed but general tightening of constraints will significantly impact placement and routing and is generally best added outside of the incremental flow.

Incremental Directives

There are 3 directives that control how the incremental flow behaves. Incremental directives are set using the command:

```
read_checkpoint -directive <directiveName> <reference>.dcp
```

RuntimeOptimized

The RuntimeOptimized directive tries to reuse as much placement and routing information from the reference run as possible. The timing target will be the same as the reference run. If the reference run has WNS -0.050, then the incremental run will not try to close timing on this design and instead also target -0.050. This impacts setup time only. This is the default behavior when no directive is specified.

TimingClosure

The `TimingClosure` directive will reuse placement and routing from the reference but it will rip up paths that do not meet timing and try to close them. Some run time intensive algorithms are run to get as much timing improvement as possible but as the placement is largely given up front gains are limited. This technique can be effective on designs with a reference WNS > -0.250 ns.

Note: For further chance of closing timing, run `report_qor_suggestions` to generate automated design enhancements.

Quick

Quick is a special mode that does not call the timer during place and route and instead uses the placement of related logic as a guide. It is the fastest mode but not applicable for most designs. Designs will need WNS > 1.000 ns to be effective. These are typically ASIC emulation or prototyping designs.

Note: In versions 2019.1 and before, the same behavior was achieved via directive mapping at place_design and route_design. The Explore directive was mapped to TimingClosure, Quick mapped to Quick and other directives mapped to RuntimeOptimized.

CAUTION! *Users upgrading from 2019.1 and earlier who are specifying the Explore or Quick directives for `place_design` will need to specify the incremental directive to achieve the equivalent functionality in 2020.1.*

Incremental Modes

The incremental implementation works in one of three modes: automatic, high reuse or low reuse.

Automatic Incremental

Automatic Incremental Implementation allows a user to activate the Incremental Implementation flow but let Vivado decide whether to use the default or incremental algorithms at the time `read_checkpoint -auto_incremental` is issued. It bases this decision on the quality of the reference checkpoint.

In order to accept the reference checkpoint, the following criteria must be met:

- 94% cell matching
- 90% net matching
- WNS > -0.250

By guaranteeing a good reference checkpoint, the incremental flow can get good QoR results and when the checkpoint is poor a new place and route solution is sought.

In project mode, the updating of the checkpoint is also managed for you and is adhering to the above criteria. In non project mode, the user has control over whether to update the checkpoint.

The flow is activated using the following command:

```
read_checkpoint -incremental -auto_incremental <reference>.dcp
```

When updating the checkpoint, it is worth checking to ensure that WNS has not degraded beyond acceptable limits. This can be done by running the following command at the end of the implementation flow:

```
if {[get_property SLACK [get_timing_path]] > -0.250} {
  file copy -force <postroute>.dcp <reference>.dcp
}
```

High Reuse Mode

In High Reuse mode, incremental implementation is run if cell reuse is above 75%. When cell reuse is below this, placement information is not reused and the flow will continue with the default algorithms. The target WNS is determined by a combination of both the reference checkpoint and the directive.

For high reuse mode, the following message is printed in the log file after place_design has started:

```
INFO: [Place 46-42] Incremental Compile is being run in High Reuse Mode.
```

Low Reuse Mode

In Low Reuse mode, reuse is determined by the `read_checkpoint -reuse_object <objects> -incremental <reference>.dcp` switch. In this mode:

- The user can target cell types, hierarchical cells, clock regions and SLRs to be reused.
- The Target WNS is always 0.
- Incremental directives are ignored and the directives from the default place and route algorithms are used.

Low reuse mode is most effective on designs that are exhibiting challenges to the place and route in specific areas. Examples of use cases are:

- Reusing Block Memory or DSP placement from a good run can improve the total number of good runs at each place and route iteration.
- Reusing a particular level of hierarchy that closes timing intermittently.

You can determine if the tool is in low reuse mode by examining the log file after `place_design` has started for the following message:

```
INFO: [Place 46-42] Incremental Compile is being run in Low Reuse Mode.
```

Running Incremental Place and Route

After the reference checkpoint is read by Vivado, the following actions are taken:

- Physical optimizations that match the ones in the reference run are carried out on the incremental design automatically.
- The netlist in the incremental 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 incremental design.

- Routing is reused to route matching nets on a per-load-pin basis. If a load pin disappears due to netlist changes, then its routing is discarded, otherwise it is reused. Therefore it is possible to have partially-reused routes.

Incremental placement and incremental routing might discard cell placements and net routes instead of reusing them, if it helps improve routability of the netlist or helps maintain performance comparable to that of the reference design.

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

read_checkpoint -incremental

After the current design is loaded, load the reference design checkpoint using the `read_checkpoint -incremental <dcp>` command. The `-incremental` option enables the Incremental Compile design flow for subsequent place and route operations

Incremental Implementation Controls

If no command arguments (other than `-incremental`) are specified, the tool reuses as much of the reference checkpoint information that it can. However, command arguments can be applied to the `read_checkpoint -incremental` command that give the user control over what is used and not reused.

-auto_incremental Option

This enables the automatic incremental flow described in Automatic Incremental.

-reuse_objects Option

```
-reuse_objects <cell objects>
```

The `-reuse_objects` can take either cells, clock regions or SLRs as an argument. When specifying cells, use the `get_cells` command. When using `get_cells`, hierarchical or leaf cells can be specified along with cell types when using the `-filter` switch. When specifying clock regions (`get_clock_regions`) or SLRs (`get_slrs`) for reuse, the cells in the region of the reference checkpoint will be reused if they exist in the incremental run. When specifying any of the arguments, net reuse is inferred based on the cells identified for reuse.

-fix_objects Option

```
-fix_objects <cell objects>
```

The `-fix_objects` option can be used to lock a subset of cells. These cells are not touched by the incremental place and route tools. The `-fix_objects` option only works on cells that match and are identified for cell reuse. This is the full design space when

`-reuse_objects` is not specified, or the associated cells when `-reuse_objects` is specified.

Examples

The following are examples of their use:

- To reuse and fix only Block Memory placement:

```
read_checkpoint -incremental routed.dcp -reuse_objects [all_rams] -fix_objects [all_rams]
```

- To reuse and fix only DSP placement:

```
read_checkpoint -incremental routed.dcp -reuse_objects [all_dsps] -fix_objects [all_dsps]
```

- To reuse both Block Memory and DSP placement, and fix the placement of all cells specified for reuse:

```
read_checkpoint -incremental routed.dcp -reuse_objects [all_rams] \
  -reuse_objects [all_dsps] -fix_reuse [current_design]
```

- To reuse all cells at and below the level of hierarchy indicated and allow the tools some flexibility to deal with changes in critical areas:

```
read_checkpoint -incremental routed.dcp \
  -reuse_objects [get_cells <cell_name>] -fix_objects [get_cells <cell_name>]
```

Using report_incremental_reuse

The `report_incremental_reuse` command is available at any stage of the flow after `read_checkpoint -incremental` has been used. The report allows the user to compare the following between the reference and current design runs:

- Examine cell, net, I/O and pin reuse in the current run
- Runtimes
- Timing WNS at each stage of the flow
- Tool options
- Tool versions
- Iphys_opt_design replaying optimization
- QoR suggestions applied with the incremental flow

By examining the cell reuse and the other factors mentioned above, a user can determine the effectiveness of the incremental. Where the flow is judged ineffective, a user would typically update the checkpoint to a newer version of the design or adjust the tool flow.

The report is split into 7 sections:

1. Flow Summary
    - This reports the general information for the current whole incremental flow:

```
1. Incremental Flow Summary
---------------------------

+-----------------------+---------------+
|    Flow Information   |     Value     |
+-----------------------+---------------+
| Synthesis Flow        |       Default |
| Auto Incremental      |            No |
| Incremental Directive | TimingClosure |
| Reuse mode            |          High |
| Target WNS            |        -0.160 |
| QoR Suggestions       |             0 |
+-----------------------+---------------+
```

2. Reuse Summary

    - This contains an overview of the cells, nets, pins, and ports that are reused.

      An example is:

```
2. Reuse Summary
----------------
+------+----------------------+--------------------+--------------------+---------+
| Type | Matched % (of Total) | Reuse % (of Total) | Fixed % (of Total) |  Total  |
+------+----------------------+--------------------+--------------------+---------+
| Cells|               100.00 |              99.82 |               0.38 |  688982 |
| Nets |                99.98 |              99.72 |               0.00 |  795869 |
| Pins |                    - |              99.11 |                  - | 2823905 |
| Ports|               100.00 |             100.00 |             100.00 |     667 |
+------+----------------------+--------------------+--------------------+---------+
```

3. Reference Checkpoint Information

    - This contains information about the reference checkpoint. From this section you can examine the:

        - Vivado version that generated it
        - Stage of the implementation
        - Recorded WNS and WHS
        - Speedfile version information of both the reference and incremental runs

      An example is:

```
3. Reference Checkpoint Information
+--------------------------------+----------------------------+
|         DCP Information        |           Value            |
+--------------------------------+----------------------------+
| Vivado Version                 |                     2020.2 |
| DCP State                      |                 POST_ROUTE |
| Recorded WNS                   |                     -0.305 |
| Recorded WHS                   |                      0.000 |
| Reference Speed File Version   | PRODUCTION 1.27 02-28-2020 |
| Incremental Speed File Version | PRODUCTION 1.27 02-28-2020 |
+--------------------------------+----------------------------+
* Recorded WNS/WHS timing numbers are estimated timing numbers. They may vary
slightly from sign-off timing numbers.
```

4. Comparison with Reference Run
   - This contains useful metrics about a comparison with the reference run. From this section you can compare:
     - Runtime information
     - WNS at each stage of the flow
     - Tool options at each stage of the flow.

   An example is:

```
4. Comparison with Reference Run
+-----------------+---------------------------+---------------------------+---------------------------+
|                 | WNS(ns)                   | Runtime(elapsed)(hh:mm)   | Runtime(cpu)(hh:mm)       |
+-----------------+-------------+-------------+-------------+-------------+-------------+-------------+
| Stage           | Reference   | Incremental | Reference   | Incremental | Reference   | Incremental |
+-----------------+-------------+-------------+-------------+-------------+-------------+-------------+
| synth_design    |             |             | 00:26       | 00:26       |             |             |
| opt_design      |             |             | 00:07       | 00:07       |             |             |
| opt_design      |             |             | 00:02       | 00:01       | 00:03       | 00:04       |
| read_checkpoint |             |             |             | 00:05       |             | 00:06       |
| place_design    | -0.769      | -0.829      | 00:12       | 00:08       | 00:23       | 00:17       |
| phys_opt_design | -0.767      | -0.829      | < 1 min     | < 1 min     | 00:01       | 00:01       |
| route_design    | -0.848      | -0.310      | 00:12       | 00:33       | 00:29       | 01:07       |
| phys_opt_design | -0.829      | -0.310      | 00:03       | 00:01       | 00:04       | 00:02       |
+-----------------+-------------+-------------+-------------+-------------+-------------+-------------+
```

Figure 2-32: **Reference Run Comparison**

5. Optimization Comparison With Reference Run
   - This section contains the iphys_opt_design replaying information which is retrieved from the reference dcp, along with the RQS suggestions derived, generated, and applied in the current incremental flow.

   An example is:

```
5.1 iphys_opt_replay Optimizations
----------------------------------

+-------------------------+--------+------------+
| iphys_opt_design replay | Reused | Not Reused |
+-------------------------+--------+------------+
| hold_fix                |    113 |          0 |
| fanout_opt              |      2 |          0 |
| critical_cell_opt       |     10 |          0 |
| restruct_opt            |      4 |          0 |
+-------------------------+--------+------------+


5.2 QoR Suggestion Optimizations
--------------------------------

+-----------------------------------------+-------+
|              QoR Suggestions            | Value |
+-----------------------------------------+-------+
| QoR Suggestions                         |     0 |
|   Suggestions Included In Reference     |     0 |
|     Yet to apply                        |     0 |
```

```
|     Applied                                |      0 |
|     Failed to apply                        |      0 |
|   New Suggestions                          |      0 |
|     Yet to apply                           |      0 |
|     Applied                                |      0 |
|     Failed to apply                        |      0 |
| Non Incremental Friendly New Suggestions   |      0 |
+--------------------------------------------+--------+
```

6. Command Comparison with Reference Run
    - This section contains the commands executed for flow command comparison.

      An example is:

```
6.1 Reference:
--------------
opt_design -directive Default
place_design -directive ExtraPostPlacementOpt
phys_opt_design -directive AlternateFlowWithRetiming
phys_opt_design -directive AggressiveFanoutOpt
phys_opt_design -directive AggressiveExplore
phys_opt_design -directive AlternateReplication
route_design -directive Explore
phys_opt_design -directive Explore

6.2 Incremental:
----------------
opt_design -directive Default
read_checkpoint -directive TimingClosure  -incremental
/group/2020.1/post_route_phys_opt2.dcp
```

7. Non-reuse Information
    - This contains metrics about what was not reused and why.

      An example is:

```
7. Non Reuse Information
------------------------
+--------------------------------------------------------+------+
|                         Type                           |  %   |
+--------------------------------------------------------+------+
| Non-Reused Cells                                       | 0.17 |
|   Discarded illegal placement due to netlist changes   | 0.17 |
|   Discarded to improve timing                          | 0.01 |
| Partially reused nets                                  | 0.00 |
| Non-Reused nets                                        | 0.27 |
| Non-Reused Ports                                       | 0.00 |
+--------------------------------------------------------+------+
```

Factors Affecting Run Time Improvement

Factors that can affect run time improvement include:

- The amount of change in timing-critical areas. If critical path placement and routing cannot be reused, more effort is required to preserve timing. Also, if the small design changes introduce new timing problems that did not exist in the reference design, higher effort and run time might be required, and the design might not meet timing.
- The initialization portion of the place and route run time. In short place and route runs, the initialization overhead of the Vivado placer and router might eliminate any gain from the incremental place and route process. For designs with longer run times, initialization becomes a small percentage of the run time.

Using Incremental Implementation

In both Project Mode and Non-Project Mode, incremental implementation 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 file name 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 follow `opt_design` and precede `place_design`.

Using Incremental Implementation in Non-Project Mode

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

1. Load the current design.
2. Run `opt_design`.
3. Run `read_checkpoint -incremental <dcp_file>`.
4. Run `place_design`.
5. Run `phys_opt_design` (optional). Run `phys_opt_design` if it was used in the reference design.
6. Run `route_design`.

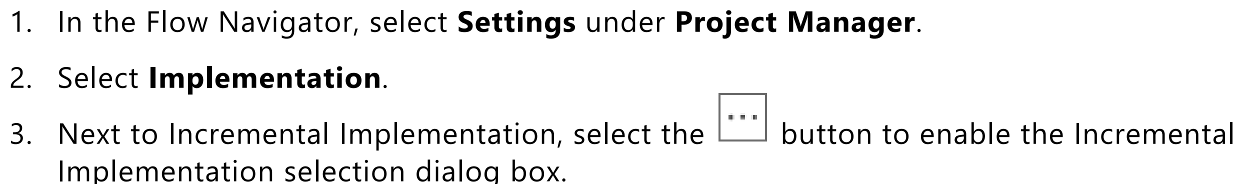
```
link_design; # to load the current design
opt_design
read_checkpoint -incremental <dcp_file>
place_design
phys_opt_design;
route_design
```

Using Incremental Implementation in Project Mode

In Project Mode, you can set the incremental compile option in two ways: in the Design Runs window and in the Implementation section of the Settings dialog box. To set the incremental compile option in the Design Runs window:

1. Right-click a run in the **Design Runs** window.
2. Click **Set Incremental Implementation** from the context menu.

To set Incremental Implementation in the **Settings** dialog box:

1. In the Flow Navigator, select **Settings** under **Project Manager**.
2. Select **Implementation**.
3. Next to Incremental Implementation, select the  button to enable the Incremental Implementation selection dialog box.

To enable automatic checkpoint management as described above in Automatic Mode...., select the **Automatically use the checkpoint from the previous run** radio button.

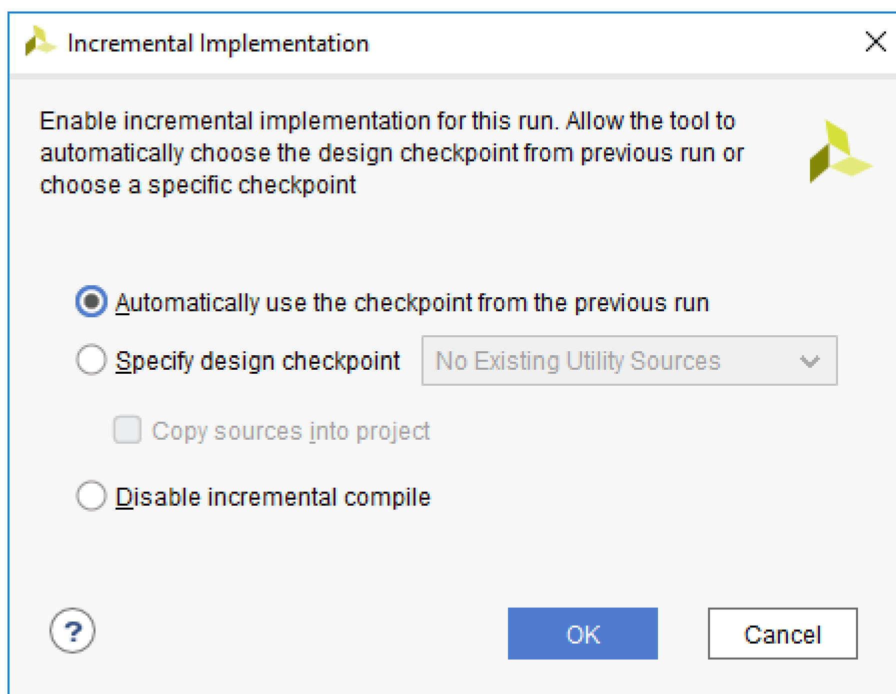


Figure 2-33: **Enabling Automatic Incremental Implementation**

Alternatively use the TCL command:

```
set_property AUTO_INCREMENTAL_CHECKPOINT 1 [get_runs <run_name>]
```

To clean the reference data, set Incremental Implementation to “Disable Incremental Compile” and reset the run. After resetting the run, it can be turned on again starting fresh.

To reference a user specified checkpoint, select the option **Specify Design Checkpoint**. When a checkpoint is selected, it will be added to the utils_1 fileset. Alternatively use the TCL commands:

```
add_files -fileset utils_1 -norecurse <reference>.dcp
set_property INCREMENTAL CHECKPOINT <reference>.dcp [get_runs <run_name>]
```

To disable incremental compile for the current run:

- Select **Disable incremental compile** in the **Incremental Implementation** window.

  or

- Run the following command in the Tcl Console:

```
set_property AUTO_INCREMENTAL_CHECKPOINT 0 [get_runs <run_name>]
```

Note: Low reuse mode is not natively supported in project mode. It can be achieved using a post opt_design Tcl script with the read_checkpoint -incremental command.

Orphaned Route Segments

Some cells might 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 might 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 Synplify Compile Points

The Incremental Compile flow is most effective when the revised and reference designs are most similar, preferably with at least 95 percent of the cells matching. Synthesis flows such as Synplify Compile Points minimize the amount of netlist changes resulting from RTL changes. Compile points are logical boundaries across which no optimization might occur. This sacrifices some design performance for predictability, but when combined with Incremental Compile, the resulting flow yields even more run time savings and predictability.

Synplify provides two different compile point flows, which are automatic and manual. In the automatic compile point mode, compile points are automatically chosen by synthesis, based on existing hierarchy and utilization estimates. This is a pushbutton mode. Aside from enabling the flow, there is no action required on your part. To enable, check the **Auto Compile Point** check box in the GUI or add the following setting to the Synplify project:

```
set_option -automatic_compile_point 1
```

The manual compile point flow offers more flexibility, but requires more interaction to choose compile points. The flow involves compiling the design, then using either the SCOPE

editor Compile Points tab or the `define_compile_point` setting. For further information on compile point flows, see the Synplify online help.

Using Incremental Synthesis

Vivado Synthesis can be run incrementally. In this flow, the tool will have a reference run that will be referred to in later runs. It will be able to detect when the design has changed and then only re-run synthesis on sections of the design that have changed. The key advantage of this flow is that for designs with small changes, the runtime will be greatly reduced. In addition, the QoR of the design will fluctuate less when small changes are inserted into the RTL.

Saving Post-Reuse Checkpoints

After `read_checkpoint -incremental` applies the reference checkpoint to the current design, the incremental reuse data is retained throughout the flow. If a checkpoint is saved, then reloaded in the same or a separate Vivado Design Suite session, it remains in incremental compile mode. Consider the following command sequence:

```
opt_design; # optimize the current design
read_checkpoint -incremental reference.dcp; # apply reference data to current design
write_checkpoint incr.dcp; # save a snapshot of the current design
read_checkpoint incr.dcp
place_design
write_checkpoint top_placed.dcp; # save incremental placement result
route_design
```

Upon `read_checkpoint incr.dcp`, the Vivado tools determine that incremental data exists, and the subsequent `place_design` and `route_design` commands run incrementally.

Even if you exit and restart the Vivado Design Suite, in the following command sequence the `route_design` command is run in incremental mode, using the routing data from the original reference checkpoint `reference.dcp`:

```
read_checkpoint top_placed.dcp
phys_opt_design
route_design
```

Constraint Conflicts

Constraints of the revised design can conflict with the physical data of the reference checkpoint. When conflicts occur, the behavior depends on the constraint used. This is illustrated in the following examples.

LOC Constraint Conflict Example

A constraint assigns a fixed location RAMB36_X0Y0 for a cell cell_A. However in the reference checkpoint reference.dcp, cell_A is placed at RAMB36_X0Y1 and a different cell cell_B is placed at RAMB36_X0Y0.

After running read_checkpoint -incremental reference.dcp, cell_A is placed at RAMB36_X0Y0 and cell_B is unplaced. The cell cell_B is placed during incremental placement.

PBlock Conflict Example

In the reference checkpoint there are no Pblocks, but one has been added to the current run. Where there is a conflict, the placement data from the reference checkpoint is used.

Incremental Compile Advanced Analysis

The Vivado tools provide reporting, timing labels, and object properties for advanced reuse analysis.

Reuse Reporting

The report_incremental_reuse command provides options for more detailed analysis, similar to report_utilization.

```
-cells <list of cells>
```

The -cells option limits the reuse reporting to the list of given cells instead of reporting reuse of the entire design.

For example, limit the reuse reporting to only block RAM:

```
report_incremental_reuse -cells [get_cells -hierarchical -filter { PRIMITIVE_TYPE =~
BLOCKRAM.*.* } ]
```

Incremental Reuse Summary

```
1. Reuse Summary
----------------

+--------+----------------------+--------------------+--------------------+-------+
|  Type  | Matched % (of Total) | Reuse % (of Total) | Fixed % (of Total) | Total |
+--------+----------------------+--------------------+--------------------+-------+
| Cells  |               100.00 |             100.00 |               0.00 |    16 |
+--------+----------------------+--------------------+--------------------+-------+


2. Reference Checkpoint Information
-----------------------------------

+----------------+------------------------+
| DCP Location:  | ./impl_1/bft_routed.dcp |
```

```
+----------------+------------------------+


+---------------------------------+-----------------------------+
|         DCP Information         |           Value             |
+---------------------------------+-----------------------------+
| Vivado Version                  |                     v2018.1 |
| DCP State                       |                  POST_ROUTE |
| Recorded WNS                    |                       1.749 |
| Recorded WHS                    |                       0.024 |
| Reference Speed File Version    | PRODUCTION 1.24.01 01-12-2017 |
| Incremental Speed File Version  | PRODUCTION 1.24.01 01-12-2017 |
+---------------------------------+-----------------------------+



3. Comparison with Reference Run
--------------------------------

+----------------+--------------------+----------------------+---------------------+
|                |       WNS(ns)      |Runtime(elapsed)(hh:mm)| Runtime(cpu)(hh:mm) |
+----------------+---------+----------+-----------+-----------+---------+----------+
|      Stage     |Reference|Incremental| Reference |Incremental|Reference|Incremental|
+----------------+---------+----------+-----------+-----------+---------+----------+
| synth_design   |    1.09 |          |   < 1 min |     00:01 |         |     00:01 |
| opt_design     |         |          |     00:01 |     00:01 |   00:01 |     00:01 |
| read_checkpoint|         |          |           |   < 1 min |         |   < 1 min |
| place_design   |   2.338 |    1.721 |   < 1 min |   < 1 min | < 1 min |   < 1 min |
| route_design   |   1.749 |    1.746 |     00:01 |     00:01 |   00:01 |    00:001 |
+----------------+---------+----------+-----------+-----------+---------+----------+

4. Non Reuse Information
------------------------

+------------------+------+
|       Type       |   %  |
+------------------+------+
| Non-Reused Cells | 0.00 |
+------------------+------+
```

Hierarchical Implementation Reuse Summary

The `-hierarchical` option displays a breakdown of cell reuse at each hierarchical level. Following is an example of `report_incremental_reuse -hierarchical`:

Note: The sample report has been truncated horizontally and vertically to fit.

```
1. Summary
----------

+--------------------------+------------------+--------+-----+---------------------+
|         Instance         |      Module      | Reused | New | Discarded(Illegal)* |
+--------------------------+------------------+--------+-----+---------------------+
| bft                      |            (top) |   3607 |   9 |                   2 |
|   (bft)                  |            (top) |    210 |   9 |                   2 |
|   arnd1                  |          round_1 |    256 |   0 |                   0 |
```

```
|      transformLoop[0].ct   | coreTransform_43 |      32 |    0 |                    0 |
|      transformLoop[1].ct   | coreTransform_38 |      32 |    0 |                    0 |
|      transformLoop[2].ct   | coreTransform_42 |      32 |    0 |                    0 |
|      transformLoop[3].ct   | coreTransform_40 |      32 |    0 |                    0 |
|      transformLoop[4].ct   | coreTransform_45 |      32 |    0 |                    0 |
+---------------------------+------------------+--------+-----+---------------------+
* Discarded illegal placement due to netlist changes
** Discarded to improve timing
*** Discarded placement by user
**** Discarded due to its control set source is unguided
***** Discarded due to its connectivity has Loc Fixed Insts
```

The reuse status of each cell is reported, beginning with the top-level hierarchy, then covering each level hierarchy contained within that level. The report progresses to the lowest level of hierarchy contained within the first submodule, then moves on to the next one.

In this example, the top level cell is `bft` with a cumulative reuse total of 3,607 cells with 9 new cells. The row with `bft` in parentheses show the cell reuse status contained within `bft` and but not its submodules. Of the 3,607 cells, only 210 are within `bft` and the remainder are within its submodules. However all 9 new cells are within `bft`. Within `bft` is a submodule `arnd1` containing 256 reused cells, and no cells within `arnd1` itself, only in submodules `transformLoop[0].ct, transformLoop[1].ct`, and so on.

There are 5 columns indicating cell reuse status at each level, although only the first one `Discarded(Illegal)` is shown. These columns have footnote references in the report with further reasons for discarding reused placement.

* Discarded illegal placement due to netlist changes

** Discarded to improve timing

*** Discarded placement by user

**** Discarded due to its control set source is unguided

***** Discarded due to its connectivity has Loc Fixed Insts

Instead of reporting all hierarchical levels, you can use the `-hierarchical_depth` option to limit the number of submodules to an exact number of levels. The following is the previous example, adding `-hierarchical_depth` of 1:

```
report_incremental_reuse -hierarchical -hierarchical_depth 1

1. Summary
----------

+----------+--------+--------+-----+---------------------+
| Instance | Module | Reused | New | Discarded(Illegal)* |
+----------+--------+--------+-----+---------------------+
|   bft    |  (top) |   3607 |   9 |                   2 |
+----------+--------+--------+-----+---------------------+
```

This limits reporting to the top level `bft`. If you had used a `-hierarchical_depth` of 2, the top and each level of hierarchy contained within `bft` would be reported, but nothing below those hierarchical cells.

Hierarchical Implementation Reuse Summary

```
1. Summary
----------

+------------------------------+--------------+--------+-----+---------------------+
|          Instance            |    Module    | Reused | New | Discarded(Illegal)* |
+------------------------------+--------------+--------+-----+---------------------+
|  bft                         |        (top) |   3607 |   9 |                   2 |
|    (bft)                     |        (top) |    210 |   9 |                   2 |
|    arnd1                     |      round_1 |    256 |   0 |                   0 |
|    arnd2                     |      round_2 |    256 |   0 |                   0 |
|    arnd3                     |      round_3 |    256 |   0 |                   0 |
|    arnd4                     |      round_4 |    256 |   0 |                   0 |
|    egressLoop[0].egressFifo  |  FifoBuffer_6 |    173 |   0 |                   0 |
+------------------------------+--------------+--------+-----+---------------------+
```

Timing Reports

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.

The following references are used with their associated meaning:

- (ROUTING): Both the cell placement and net routing are reused.
- (PLACEMENT): The cell placement is reused but the routing to the pin is not reused.
- (MOVED): Neither the cell placement nor the routing to the pin is reused.
- (NEW): The pin, cell, or net is a new design object, not present in the reference design.

See the following example.

```
---------------------------------------------------------------------------------
--------------
Routing  SLICE_X65Y175  FDRE(Prop_EFF_SLICEL_C_Q)
                                     0.114 -0.446 r  base_mb_i/microblaze_0/Q
                   net (fo=637, routed)  0.752 0.306
   base_mb_i/microblaze_0/reset_bool_for_rst
Routing  SLICE_X73Y171  FDRE                                 r
base_mb_i/microblaze_0/command_reg_clear_reg/R
------------------------------------------------------------------------------
-------------------
```

The above report, as it appears in the Vivado IDE, appears below.

Data Path

Delay Type	Incr (ns)	Path (...	Location	Netlist Resource(s)	Pin Re...
FDRE (Prop_EFF_SLICEL_C_Q)	(r) 0.114	-0.446	Site: SLICE_X65Y175	base_mb_i/microblaze_0/U...c_Reset.sync_reset_r...	Routing
net (fo=637, routed)	0.752	0.306		base_mb_i/microblaze_0/U..._Perf/reset_bool_for...	
FDRE			Site: SLICE_X73Y171	base_mb_i/microblaze_0/...ommand_reg_clear_re...	Routing
Arrival Time		0.306			

Figure 2-34: **Incremental Reuse Summary in Vivado**

To remove the labels from the timing report, use the `report_timing -no_reused_label` option.

Object Properties

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 on cell, port, net, and pin objects. The property is set to TRUE on the respective object if any of the following incremental data is reused:
  - A cell placement
  - A package pin assignment for a port
  - Any portion of the routing for a net
  - Routing to a pin
- `REUSE_STATUS`: A string property on cells and nets denoting the reuse status after incremental placement and routing.

  Possible values for cells are:
  - New
  - Reused
  - Discarded placement to improve timing
  - Discarded illegal placement due to netlist changes

  Possible values for nets are:
  - `REUSED`
  - `NON_REUSED`
  - `PARTIALLY_REUSED`
- `IS_MATCHED`: A boolean property assigned to a primitive-level cell. The property is set to `TRUE` on leaf cells that have matching leaf cells in the reference design. Matching cells are eligible for placement reuse.

TIP: *Xilinx has published several applications in XHUB, in the Incremental Compile package. These applications include visualization of placement and routing reuse when analyzing critical path and*

other design views. Also included is an application for automatic Incremental Compile for the project flow, which automatically manages reference checkpoints for incremental design runs.

TIP: *For more information on how to effectively use incremental compile, see this link in UltraFast Design Methodology Guide for the Vivado Design Suite (UG949) [Ref 13].*

Analyzing and Viewing Implementation Results

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.

- You can find a run status indicator in the project status bar at the upper right corner of the Vivado® IDE, as shown in Figure 3-1. The run status indicator displays a scrolling bar to indicate that the run is in process. You can click **Cancel** to end the run.

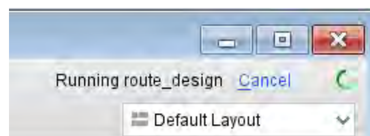


Figure 3-1: **Run Status Indicator**

- You can also find a run status indicator in the Design Runs window, as shown at the bottom left of Figure 3-2. It displays a circular arrow (noted in red in the figure) to indicate that the run is in process. You can select the run and use the **Reset Run** command from the popup menu to cancel the run.

Tcl Console | Messages | Log | Reports | Design Runs

Name	Part	Constraints	Strategy	Status	Progress
synth_1	xc7k70tfbg484-2	constrs_1	Vivado Synthesis Defaults (Vivado Synthesis 2017)	synth_design Complete!	100%
impl_1	xc7k70tfbg484-2	constrs_1	Vivado Implementation Defaults (Vivado Implementation 2017)	Running route_design...	75%

Figure 3-2: **Implementation Run Status**

Cancelling or Resetting the Run

If you cancel a run that is in-progress, by clicking either **Cancel** or **Reset Run**, the Vivado IDE prompts you to delete any run files created during the canceled run, as shown in Figure 3-3.

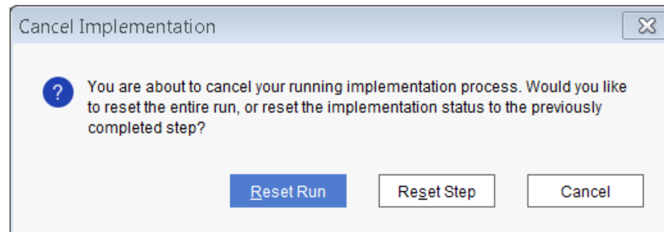


Figure 3-3: **Cancel Implementation**

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, shown Figure 3-4, can help you understand where different messages originate to aid in debugging the implementation run.

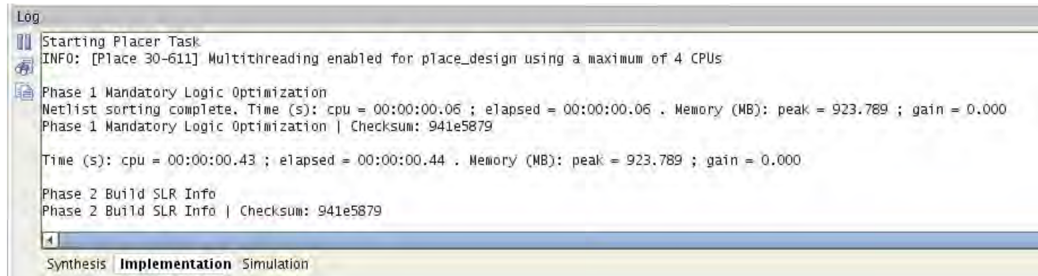
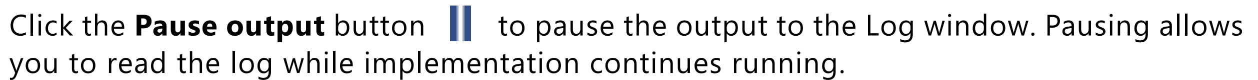


Figure 3-4: **Log Window**

Pausing Output

Click the **Pause output** button  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**, as shown in Figure 3-5.

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 might be necessary to rerun implementation, or both synthesis and implementation.

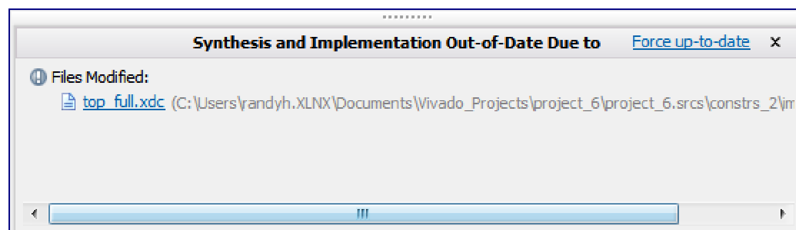


Figure 3-5: **Implementation Out-of-Date**

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.*

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 this link in the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 10].

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, as shown in Figure 3-6.

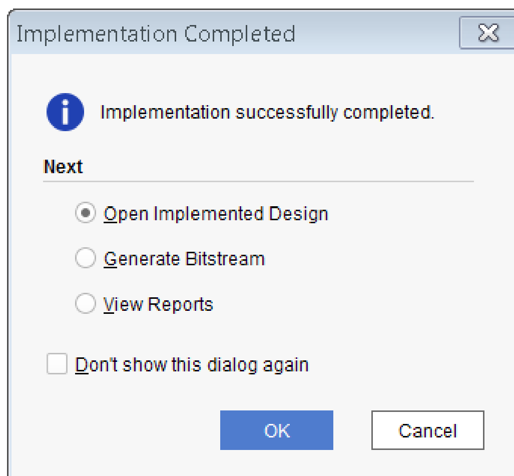


Figure 3-6: **Project Mode - Implementation Completed**

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 this link in the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 12].
   - **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 138.
2. Click **OK**.

Viewing Messages

IMPORTANT: *Review all messages. The messages might suggest ways to improve your design for performance, power, area, and routing. Critical warnings might 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, shown in Figure 3-7, 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.

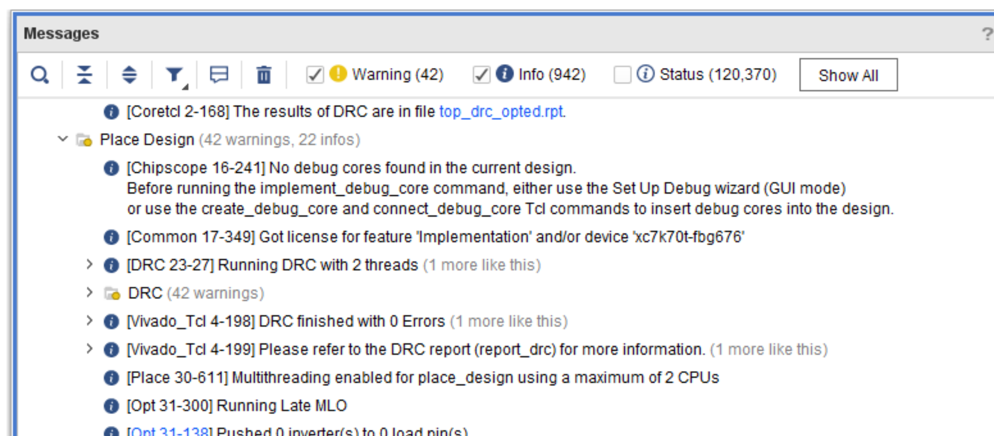


Figure 3-7: **Messages Window**

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 tools log file reports incremental placement and routing summary results from Incremental Compile.

Incremental Placement Summary

The following example of the Incremental Placement Summary includes a final assessment of cell placement reuse and run time statistics.

```
+--------------------------------------------------------------------------------+
|Incremental Placement Summary                                                    |
+--------------------------------------------------------------------------------+
|                            Type                         | Count  | Percentage |
+--------------------------------------------------------------------------------+
|  Total instances                                        | 33406 |     100.00 |
|  Reused instances                                       | 32390 |      96.96 |
|  Non-reused instances                                   |  1016 |       3.04 |
|    New                                                  |   937 |       2.80 |
|    Discarded illegal placement due to netlist changes   |    16 |       0.05 |
|    Discarded to improve timing                          |    63 |       0.19 |
+--------------------------------------------------------------------------------+
|Incremental Placement Runtime Summary                                            |
+--------------------------------------------------------------------------------+
|  Initialization time(elapsed secs)                                   |  79.99 |
|  Incremental Placer time(elapsed secs)                               |  31.19 |
+--------------------------------------------------------------------------------+
```

Incremental Routing Summary

The Incremental Routing Summary displays reuse statistics for all nets in the design. The categories reported include:

- **Fully Reused**

  The entire routing for a net is reused from the reference design.

- **Partially Reused**

  Some of the routing for a net 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                      |
---------------------------------------------------------
|Type                      | Count      | Percentage    |
---------------------------------------------------------
|Fully reused nets         |      30393|          96.73 |
|Partially reused nets     |          0|           0.00 |
|Non-reused nets           |       1028|           3.27 |
---------------------------------------------------------
```

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.

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 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 3-9 shows an example of a report opened in a Vivado IDE window.

TIP: *The directory to which the reports are to be written 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 19].

Reporting in Project Mode

In Project Mode, many reports are generated automatically. View report files in the Reports window, shown in Figure 3-8.

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

- Select the **Reports** link in the Project Summary.
- 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, page 29*.

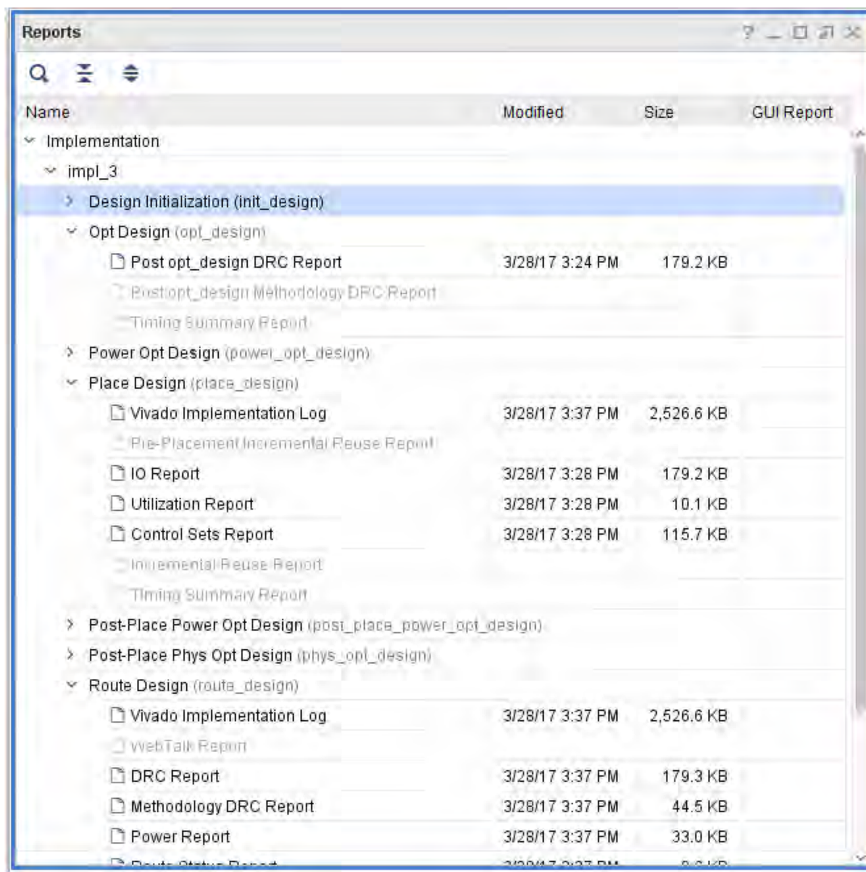


Figure 3-8: **Example Reports View**

The reports available from the Reports window contain information related to the run. The selected report opens in text form in the Vivado IDE, as shown in Figure 3-9.

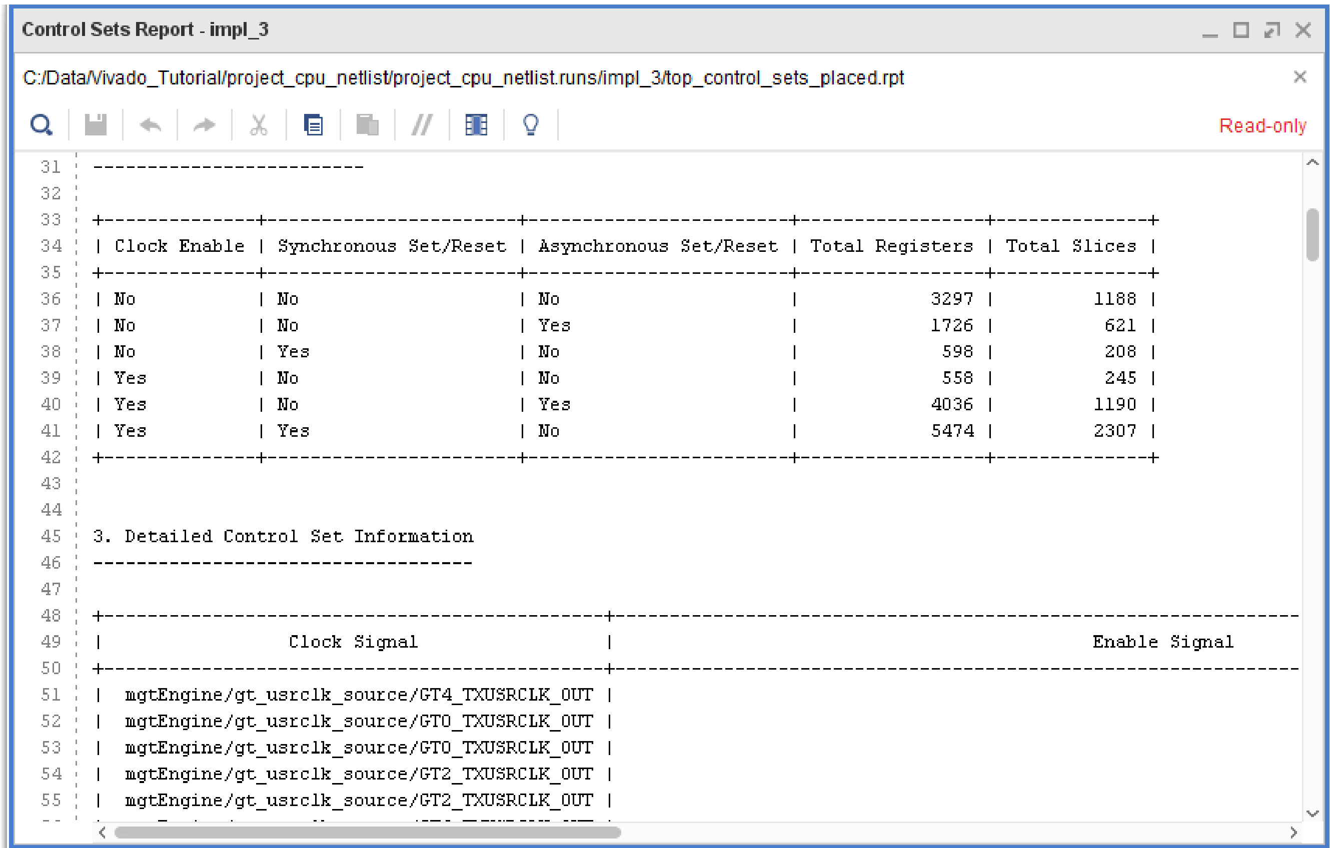


Figure 3-9: **Control Sets Report**

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 (as shown in Figure 3-8).

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 3-10 shows an example of cross probing between the Timing Summary report and the Device window. The following steps take place in this Non-Project Mode example:

- A post-route design checkpoint is opened in the Vivado IDE.
- The Timing Summary report is generated and opened using `report_timing_summary -name`.
- The Routing Resources are enabled in the Device window.
- When the timing path is selected in the Timing Summary report, cross probing on the path occurs automatically in the Device window, as shown in Figure 3-10.

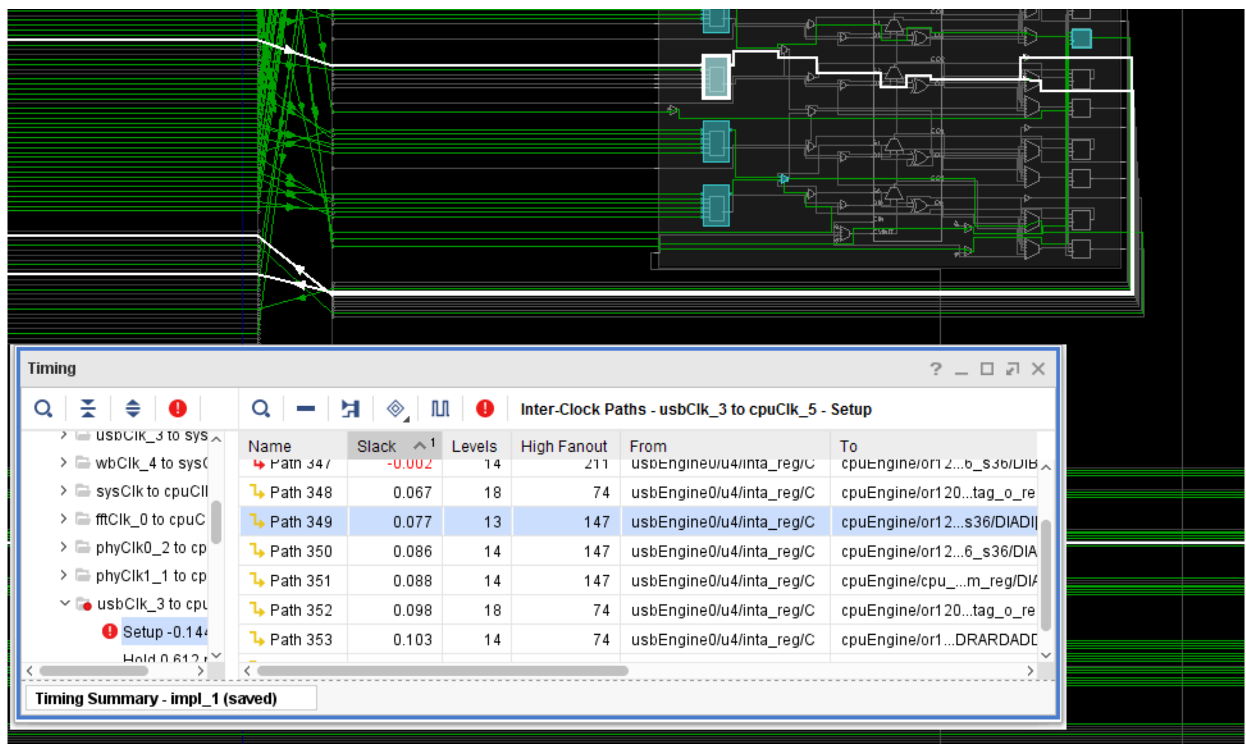


Figure 3-10: **Cross-Probing Between Timing Report and Device Window**

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 10].

Modifying Implementation Results

This section describes how to modify placement, routing, and logic for your design.

Modifying Placement

The Vivado tools track two states for placed cells, Fixed and Unfixed, which describes the way in which the Vivado tools view placed cells in the design.

Fixed Cells

Fixed cells are those that you have placed yourself, or the location constraints for the cells have been imported from an XDC file.

- The Vivado Design Suite treats these placed cells as Fixed.
- Fixed cells are not moved unless directed to do so.
- The FF in Figure 3-11 is shown in orange (default) to indicate that it is Fixed.

Unfixed Cells

Unfixed cells have been placed by the Vivado tools in implementation, during the `place_design` command, or on execution of one of optimization commands.

- The Vivado Design Suite treats these placed cells as Unfixed (or loosely placed).
- These cells can be moved by the implementation tools as needed in design iterations.
- The LUT in Figure 3-11 is shown in blue (default) to indicate that it is Unfixed.

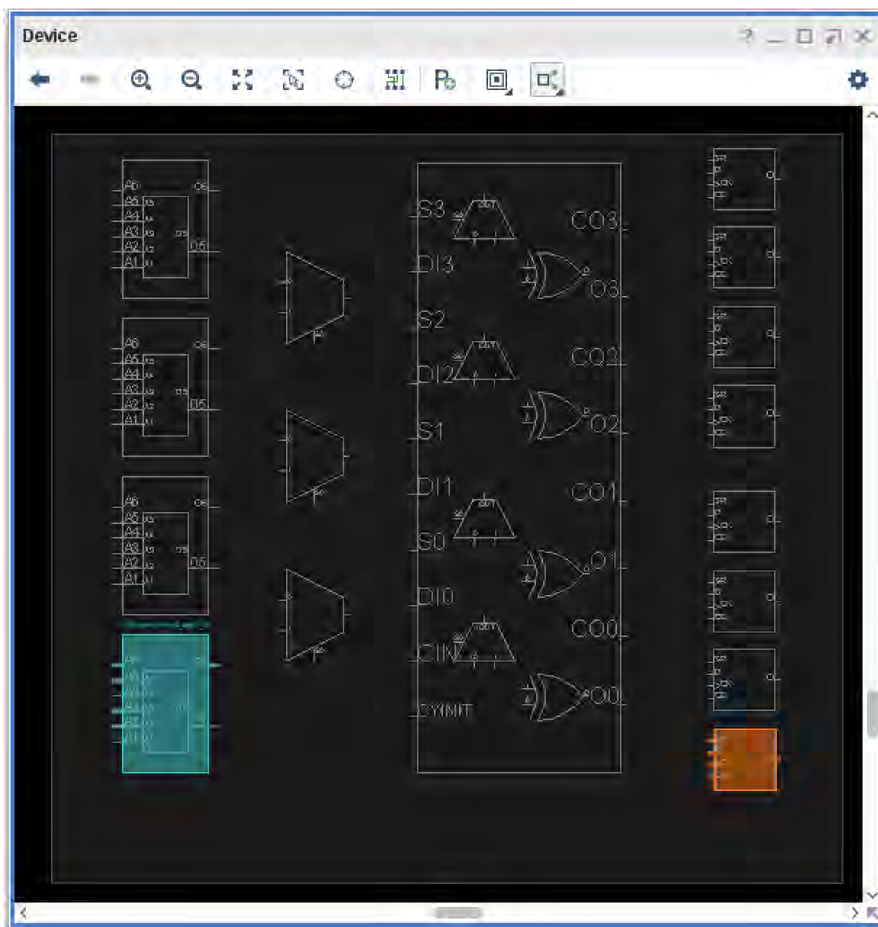


Figure 3-11: **Logic Placed in a Slice**

Both LOCS and BELS can be fixed. The placement above generates the following constraints:

```
set_property is_bel_fixed true [get_cells [list {usbEngine0/u4/u6/csr0_reg[6]}]]
set_property is_loc_fixed true [get_cells [list {usbEngine0/u4/u6/csr0_reg[6]}]]
```

There is no placement constraint on the LUT. Its placement is unfixed, indicating that the placement should not go into the XDC.

Fixing Placer-Placed Logic

To fix cells placed by the Vivado placer in the Vivado IDE:

1. Select the cells.
2. Choose **Fix Cells** from the popup menu.

To fix cell placement with Tcl, use a command of this form:

```
set_property is_bel_fixed TRUE [get_cells [list {fftEngine/control_reg_reg[1]_i_1}]]
set_property is_loc_fixed TRUE [get_cells [list {fftEngine/control_reg_reg[1]_i_1}]]
```

For more information on Tcl commands, see the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 19], or type `<command> -help`.

Placing and Moving Logic by Hand

You can place and move logic by hand.

- If the cell is already placed, drag and drop it to a new location.
- If the cell is unplaced:
  a. Enter **Create BEL Constraint Instance Drag & Drop** mode.
  b. Drag the logic from the Netlist window, or from the Timing Report window, onto the Device window.

  The logic snaps to a new legal location.

TIP: *When dragging logic to a location in the Device Window, the GUI allows you to drop the logic only on legal locations. If the location is illegal (for example, because of control set restriction for Slice FFs), the logic does not "snap" to the new location in the Device view, and it cannot be assigned.*

Hand-placing logic can be slow, and used in specific situations only. The constraints are fragile with respect to design changes because the cell name is used in the constraint.

Placing Logic using a Tcl Command

You can place logic onto device resources of the target part using the `place_cell` Tcl command. Cells can be placed onto specific BEL sites (for example, SLICE_X49Y60/A6LUT) or into available sites (for example, SLICE_X49Y60). If you specify the site but not the BEL, the tool determines an appropriate BEL within the specified site if one is available. You can use the `place_cell` command to place cells or to move placed cells from one site on the device to another site. The command syntax is the same for placing an unplaced cell or for moving a placed cell.

TIP: *When assigning logic to an illegal location (for example, because of control set restriction for Slice FFs), the Tcl Console issues an error message, and the assignment is ignored.*

Cells that have been placed using the `place_cell` Tcl command are treated as *Fixed* by the Vivado tool.

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.

To Unroute, Route, or Fix Routing on a net:

1. Open Device View.
2. Select the net.
    - Unrouted nets are indicated by a red flyline
    - Partially routed nets are highlighted in yellow
    - Nets with fixed routing are indicated by a dashed route
3. Right-click and select **Unroute**, **Route**, or **Fix Routing**.

- **Unroute** and **Route**: Calls the router in re-entrant mode to perform the operation on the net. For more information, see route_design in Chapter 2.
- **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 to route a net manually. For more information, see Manual Routing, below.

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

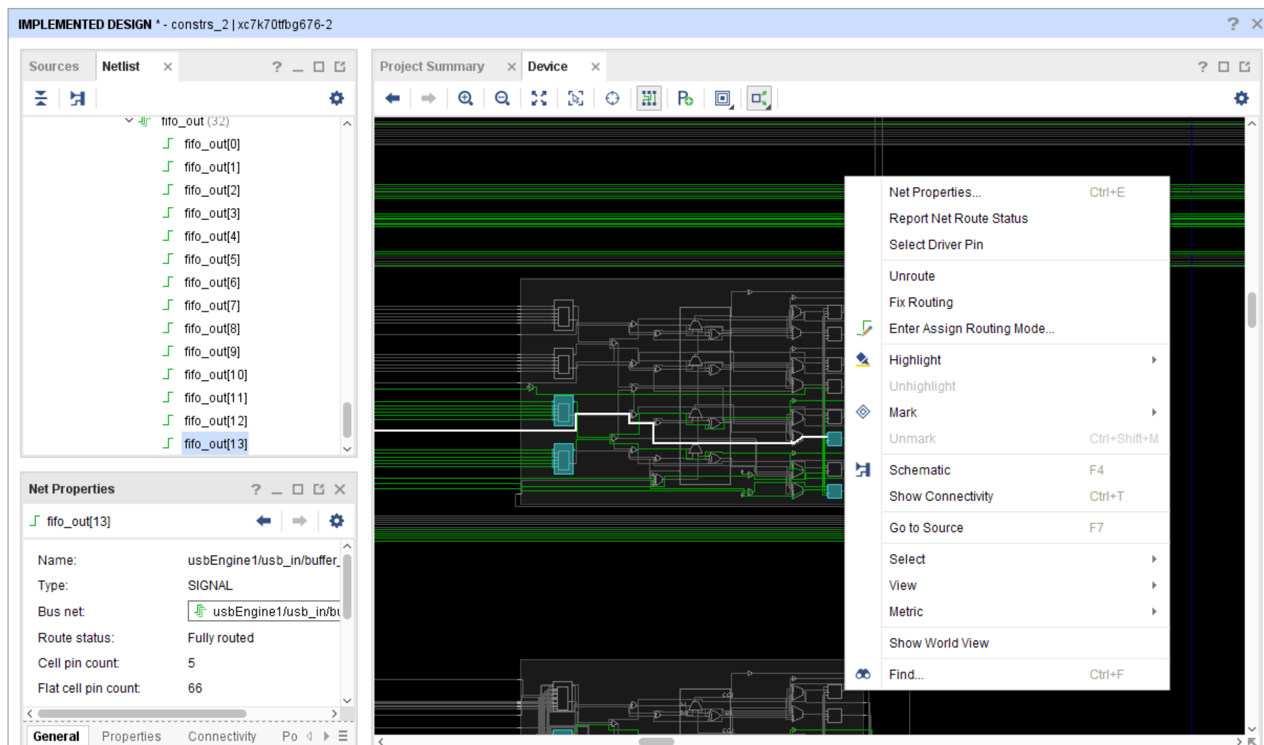


Figure 3-12: **Modify Routing**

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 might 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 the device 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. Be sure that **Routing Resources** in the Device window is selected.
3. Enable the Layers for **Unrouted Net** and **Partially Routed Net** in the Device Options Layers view, shown in Figure 3-13.

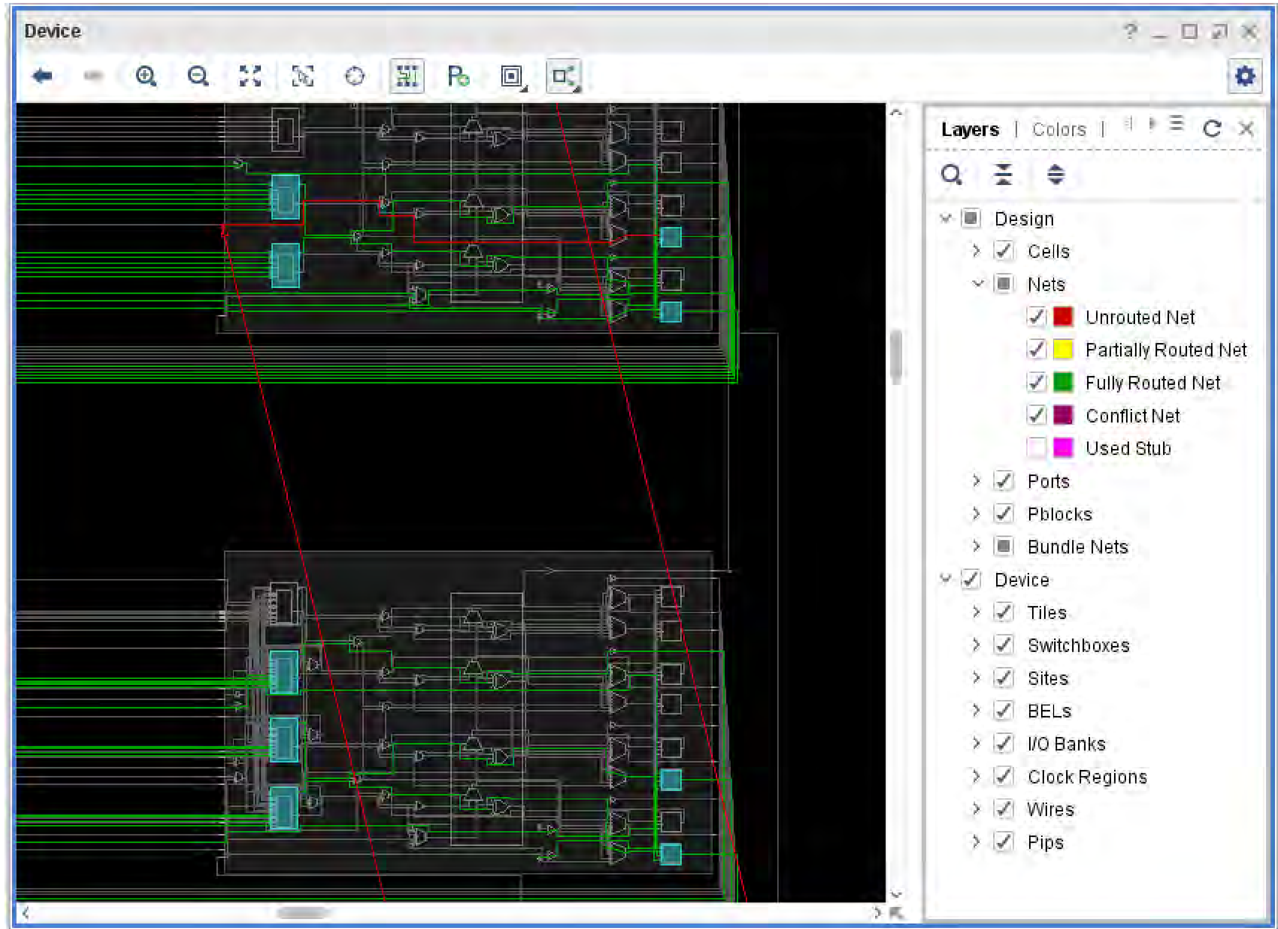


Figure 3-13: **Device Options Layers**

4. Select the net that requires routing.
   - Unrouted nets are indicated by a red flyline.
   - Partially routed nets are highlighted in yellow.
5. Right-click and select **Enter Assign Routing Mode**.

   The Target Load Cell Pin window opens.
6. Optionally, select a load cell pin to which you want to route.
7. Click **OK**.

Note: To display partially routed or unrouted nets in the Device View, ensure that those layers are selected in the Device Options menu, shown in Figure 3-14.

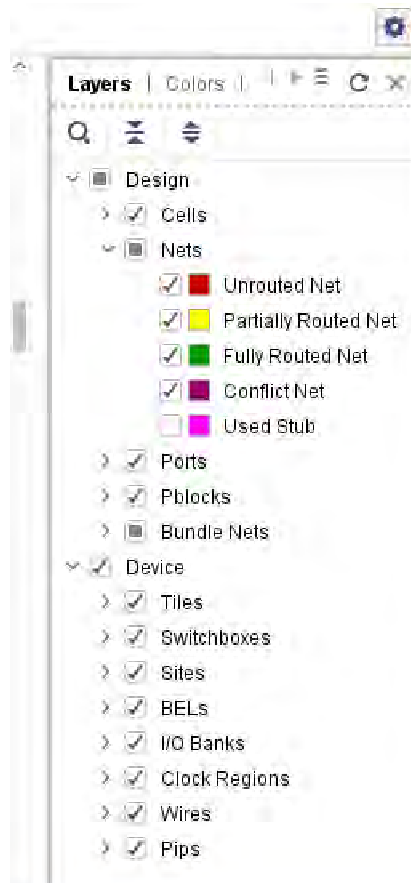


Figure 3-14: **Device Options Pull-Out Menu**

You are now in Manual Routing Mode. A Routing Assignment window, shown in Figure 3-15, appears next to the Device View.

Routing Assignment Window

Figure 3-15: **Routing Assignment Window**

The Routing Assignment window is divided into the Options, Assigned Nodes, and Neighbor Nodes sections:

- The Options section, shown in Figure 3-16, controls the settings for the Routing Assignment window.

Figure 3-16: **Routing Assignment Options**

  - The **Number of hops** value allows you to specify the number of routing hops that can be assigned for Neighbor Nodes. This also affects the Neighbor Nodes

displayed. If the number of hops is greater than 1, only the last node of the route is displayed in the Neighbor Nodes section.

- The **Maximum number of neighbors** value allows you to limit the number of neighbor nodes that are displayed in the Neighbor Nodes section. Only the last node of the route is displayed.
- The **Allow overlap with unfixed nets** switch controls whether overlaps of assigned routing with existing unfixed routing is allowed. Any overlaps need to be resolved by running the `route_design` command after fixed route assignment.

The Options section is hidden by default. To show the Options section, click **Show**.

- 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 auto-route gaps:

  a. Right-click a net gap in the Assigned Nodes section.

  b. Select **Auto-route** from the context-sensitive menu.

  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.

- The Neighbor Nodes section (shown in Figure 3-17) displays the allowed neighbor nodes, highlights the current selected nodes (in white). and highlights the allowed neighbor nodes (white dotted) in the Device View.

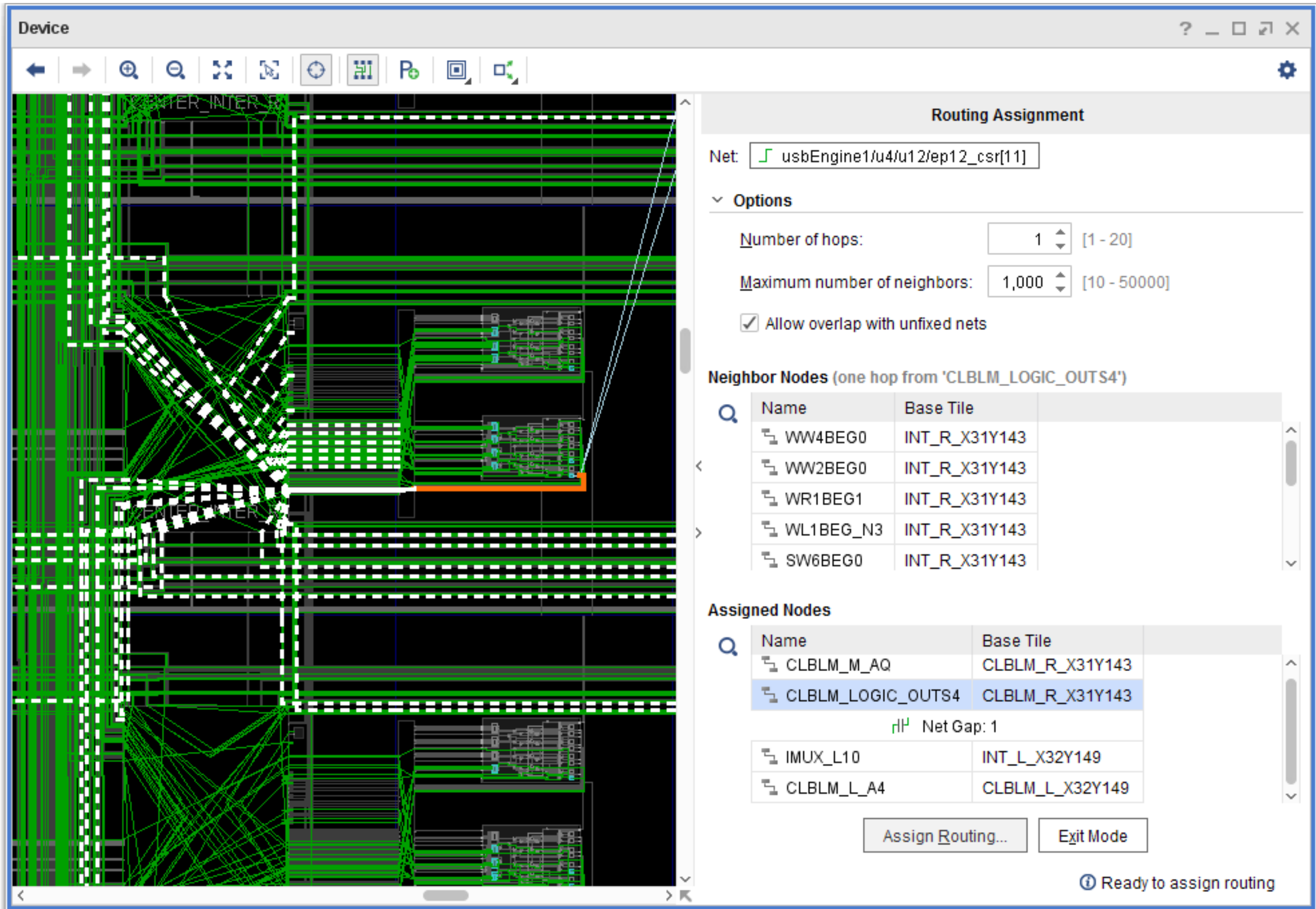


Figure 3-17: **Assign Next Routing Segment**

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.
- Click the node in the Device View

After 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, click the **Assign Routing** button in the Routing Assignment window.

The Assign Routing Window is displayed, as shown in Figure 3-18, allowing you to verify the assigned nodes before they are committed.

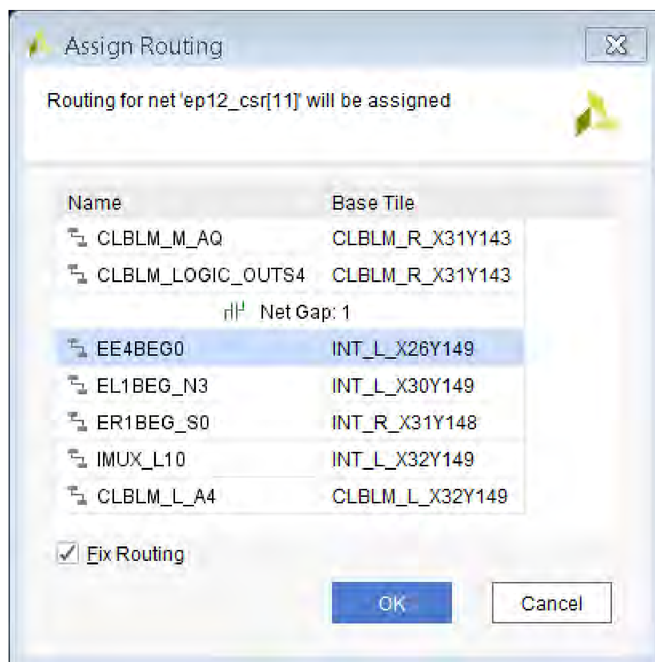


Figure 3-18: **Assign Routing Confirmation**

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.

Figure 3-19 shows an example of an assigned and partially assigned route.

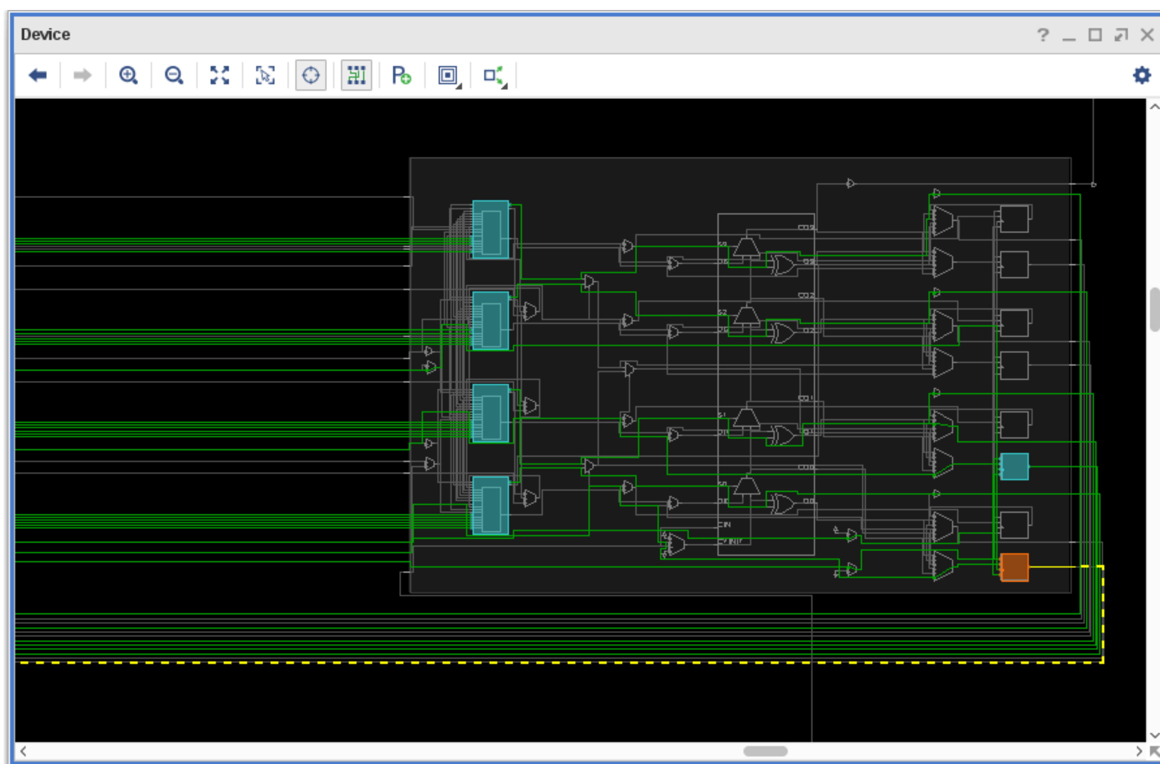


Figure 3-19: **Assigned Partially Assigned Routing**

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 provided in Entering Assign Routing Mode, page 147, above.
2. Assign routing to all the branches of the net.

Figure 3-20 shows an example of a net that has assigned routing to one load and requires routing to two additional loads.

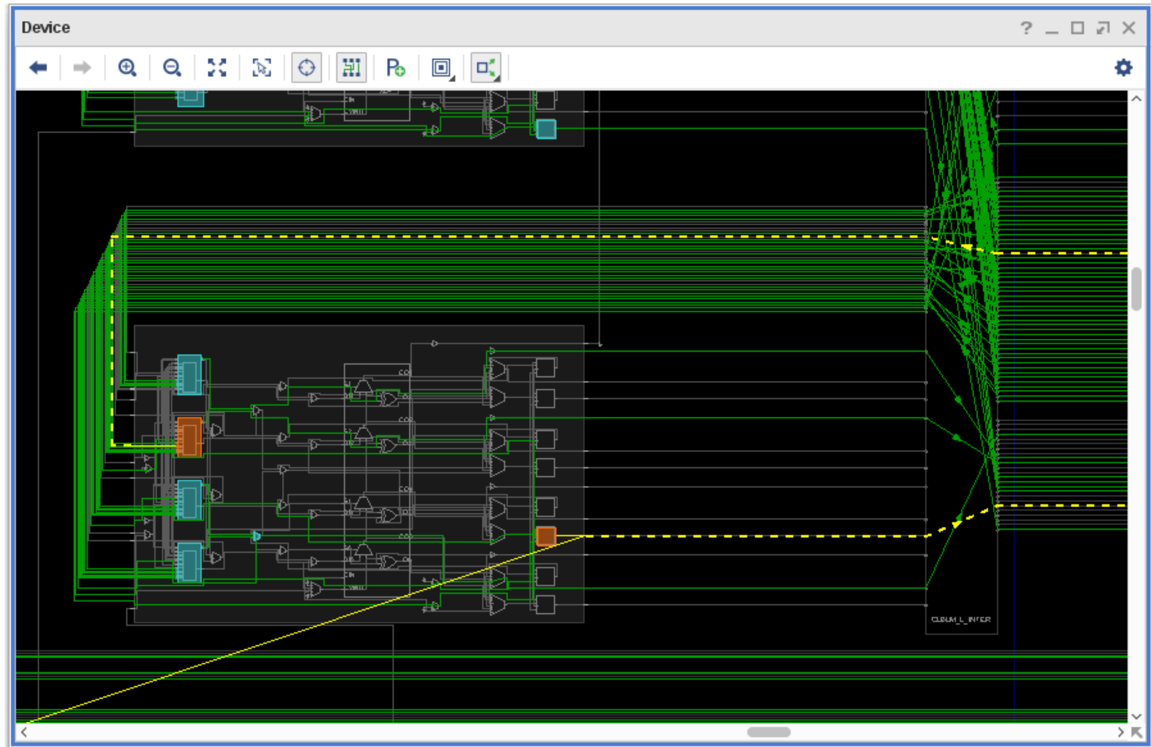


Figure 3-20: **Assign Branching Route**

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 and 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.

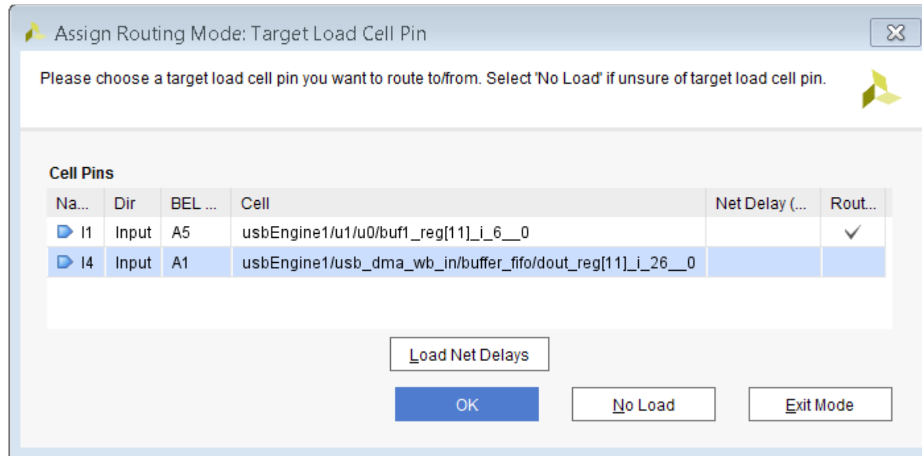


Figure 3-21: **Target Load Cell Pin (Multiple Loads)**

4. Select the load to which you want to route.
5. Click **OK**. The Branch Start window (shown in Figure 3-22) opens.

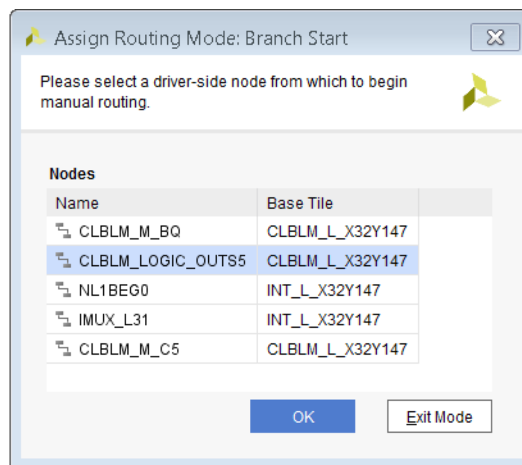


Figure 3-22: **Branch Start**

6. Select the node from which you want to branch off the route for your selected load.
7. Click **OK**.
8. Follow the steps shown in Assigning Routing Nodes, page 152.

Locking Cell Inputs and Adding DONT_TOUCH Constraint 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**.

The equivalent Tcl command is:

```
set_property LOCK_PINS {NAME:BEL_PIN} <cell object>
```

To prevent pin swapping in Physical Synthesis in the Placer, a DONT_TOUCH constraint needs to be applied to the LUT cell. The TCL command is:

```
set_property DONT_TOUCH TRUE <cell object>
```

For nets that have fixed routing and multiple LUT loads, the following Tcl script can be used to lock the cell inputs of all the LUT loads.

```
set fixed_nets [get_nets -hierarchical -filter IS_ROUTE_FIXED]
foreach LUT_load_pin [get_pins -leaf -of [get_nets $fixed_nets] \
  -filter DIRECTION==IN&&REF_NAME=~LUT*] {
    set pin [get_property REF_PIN_NAME $LUT_load_pin]
    set BEL_pin [file tail [get_bel_pins -of [get_pins $LUT_load_pin]]]
    set LUT_name [get_property PARENT_CELL $LUT_load_pin]
    # need to handle condition when LOCK_pins property already exists on LUT
    set existing_LOCK_PIN [get_property LOCK_PINS [get_cells $LUT_name]]
    if { $existing_LOCK_PIN ne "" } {
            reset_property LOCK_PINS [get_cells $LUT_name]
    }
    set_property LOCK_PINS \
      [lsort -unique [concat $existing_LOCK_PIN $pin:$BEL_pin]] [get_cells $LUT_name]
}
```

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 3-23, below. In this simplified illustration 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 is as follows:

```
{A B { D E T } 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.

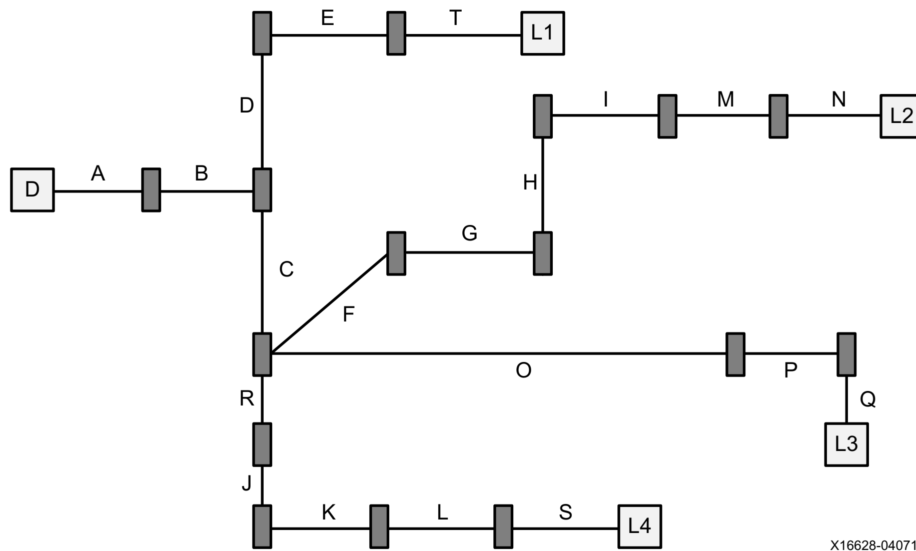


Figure 3-23: **Branch Route Example**

Using the find_routing_path Command to Create Directed Routing Constraints

The `find_routing_path` Tcl command can be used to create directed routing constraints. You can then assign the created constraints to the `FIXED_ROUTE` property of a net to lock down the routing.

For partially routed nets, the nodes can be found associated directly to the net. Refer to the *Vivado Design Suite Properties Reference Guide* (UG912) [Ref 14] for more information on the relationship between these objects.

The `find_routing_path` command returns one of the following:

- A list of nodes representing the route path found from the start point to the end point
- `no path found` if the command runs but has no result
- An error if the command fails to run.

Modifying Logic

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

Note: For more information about Tcl commands, see the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 19], 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.

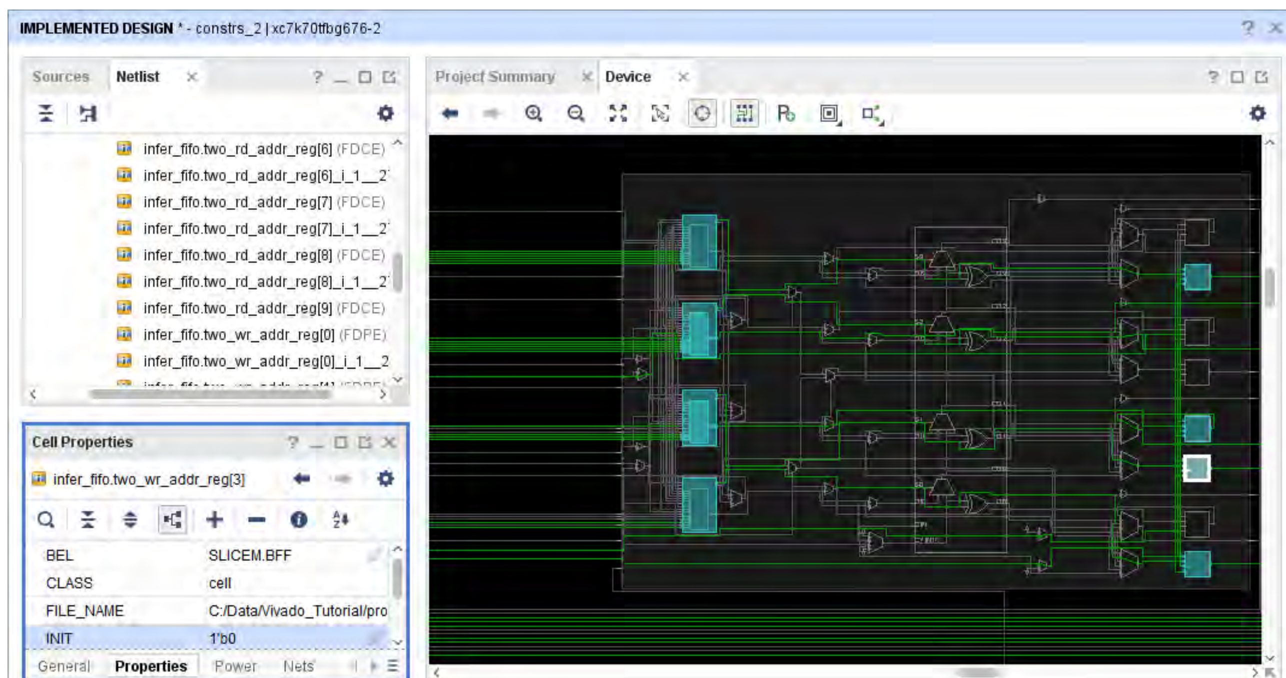


Figure 3-24: **Property Modify**

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 tools determine the appropriate INIT.

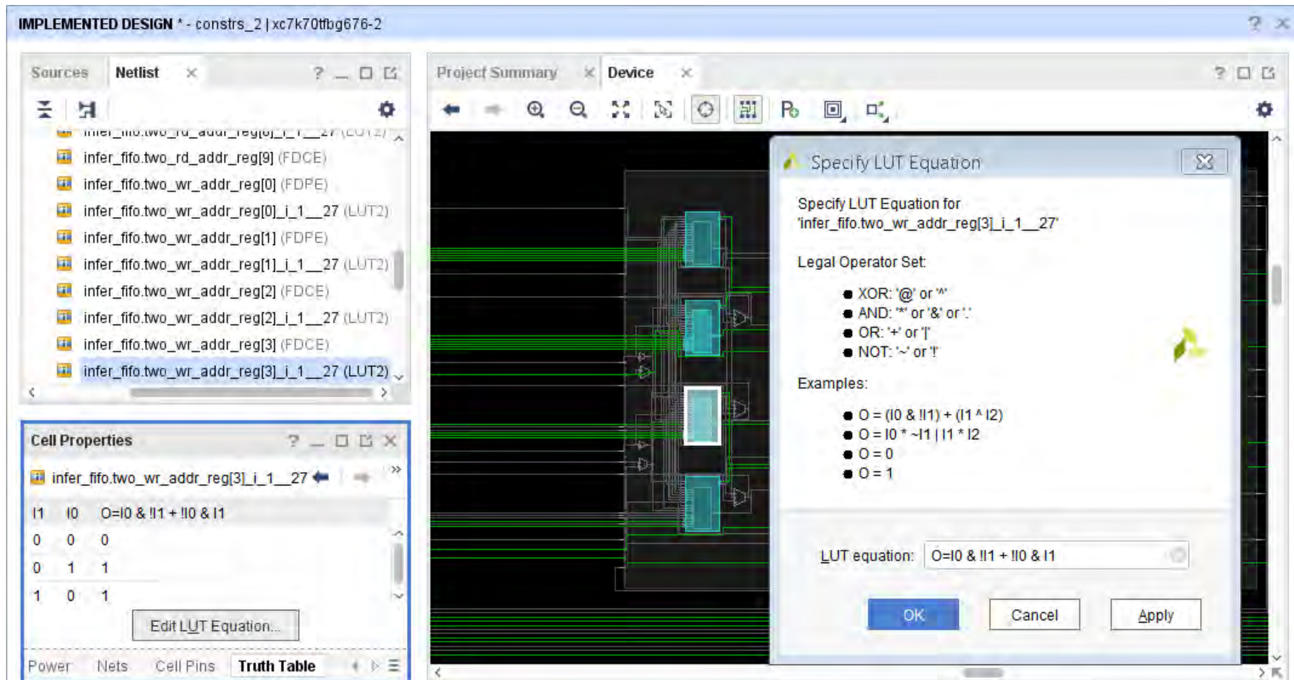


Figure 3-25: **Equation Editor**

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 > Constraints > Save**.

Modifying the Netlist

Netlists sometimes require changes to fix functional logic bugs, meet timing closure, or insert debug logic. You can modify an existing netlist using Tcl commands post-synthesis, post-place, and post-route.

Netlist Modifying Commands

The following commands allow you to modify an existing netlist:

- create_port
- remove_port
- create_cell

- remove_cell
- create_pin
- remove_pin
- create_net
- remove_net
- connect_net
- disconnect_net

Note: For more information about these Tcl commands, see the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 19], or type `<command> -help`.

The netlist modifying commands work on a post-synthesis, post-place or post-route netlist. Before the netlist is modified, it must be loaded into memory. The netlist modifying commands allow you to make logical changes to the netlist when it is in memory. You can use the `write_checkpoint` command to save changes.

TIP: *The Vivado tools allows you to make netlist changes unconditionally using the netlist modifying commands. However, logical changes can lead to invalid physical implementation. It is recommended to run DRCs after performing your netlist changes. In addition, DRCs are run as part of the process of adding the logical changes to the physical implementation. These DRCs flag any invalid netlist changes or new physical restrictions that need to be addressed before physical implementation can commence.*

Logical changes are reflected in the schematic view as soon as the netlist modifying commands are executed. Figure 3-26 shows an example of a cell that was created using a `LUT1` as a reference cell.

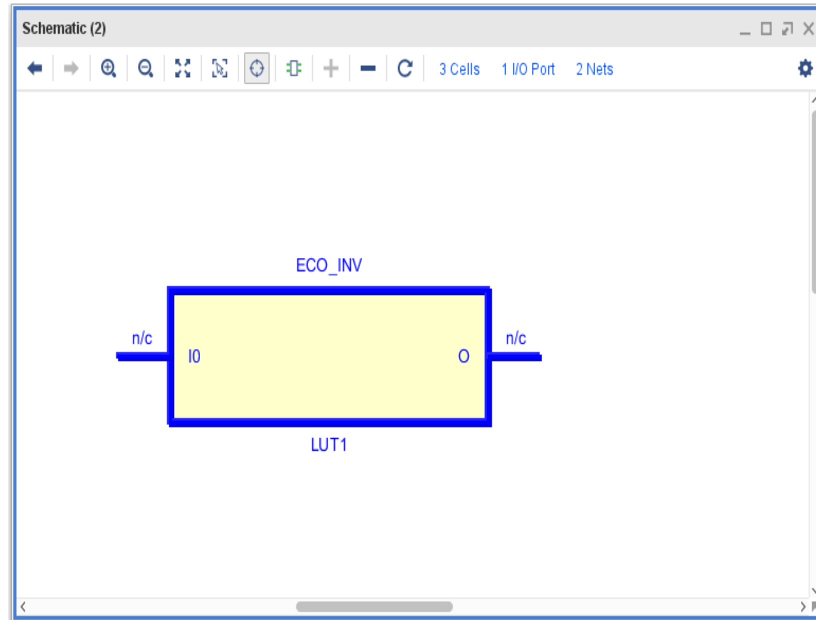


Figure 3-26: **Cell Created Using LUT1 as a Reference Cell**

When the output of the `LUT1` is connected to an `OBUF`, the schematic reflects this change showing the `ECO_INV/O` pin no longer with a "no-connect". Figure 3-27 shows the resulting schematic view.

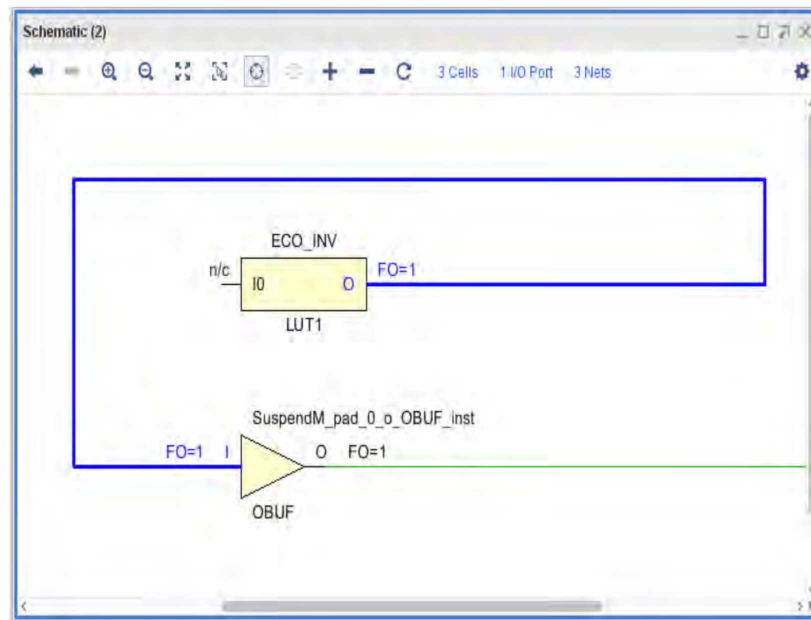


Figure 3-27: **Schematic After Connection of LUT1 to an OBUF**

Use Cases

The following examples show some of the most common use cases for netlist modifications. The examples show the schematic of the original logical netlist, list the netlist modifying Tcl commands, and show the schematic of the resulting modified netlist.

Use Case 1: Inverting the Logical Value of a Net

Inverting the logical value of a net can be as simple as modifying the existing LUT equations of a LUTx primitive, or it can require inserting a `LUT1` that is configured to invert the output from its input. The schematic in Figure 3-28 shows a `FDRE` primitive that is driving the output port `wbOutputData[0]` through an `OBUF`.

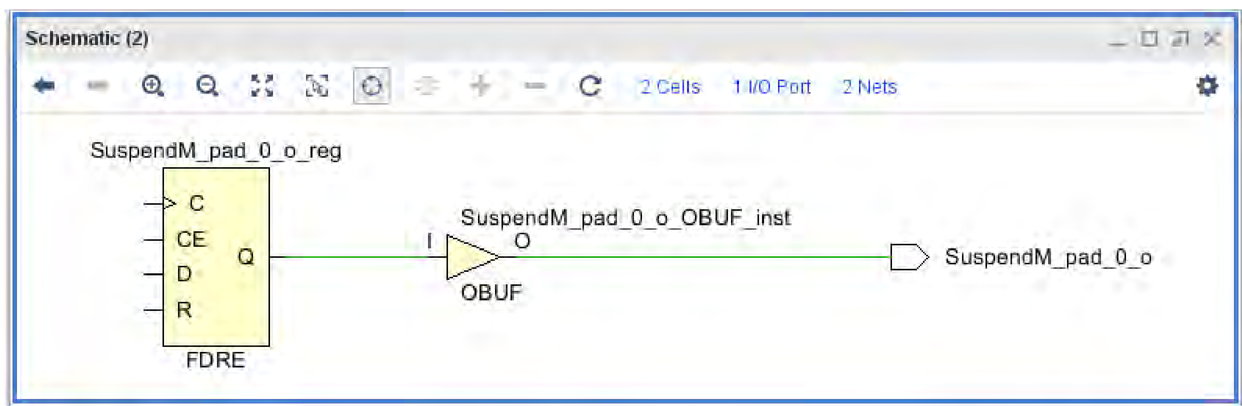


Figure 3-28: **FDRE Primitive Driving Output Port through an OBUF**

The following Tcl commands show how to add an inverter between the output of the `FDRE` and the `OBUF`:

```
create_cell -reference LUT1 ECO_INV
set_property INIT 2'h1 [get_cells ECO_INV]
disconnect_net -net {n_0_SuspendM_pad_0_o_reg} -objects \
  [get_pins {SuspendM_pad_0_o_reg/Q}]
connect_net -net {n_0_SuspendM_pad_0_o_reg} -objects [get_pins {ECO_INV/O}]
create_net ECO_INV_in
connect_net -net ECO_INV_in -objects [get_pins {SuspendM_pad_0_o_reg/Q ECO_INV/I0}]
```

In this example script, LUT1 cell `ECO_INV` is created, and the `INIT` value is set to `2'h1`, which implements an inversion. The net between the `FDRE` and `OBUF` is disconnected from the `Q` output pin of the `FDRE`, and the output of the inverting LUT1 cell `ECO_INV` is connected to the `I` input pin of the `OBUF`. Finally, a net is created and connected between the `Q` output pin of the `FDRE` and the `I0` input pin of the inverting `LUT1` cell.

Figure 3-29 shows the schematic of the resulting logical netlist changes.

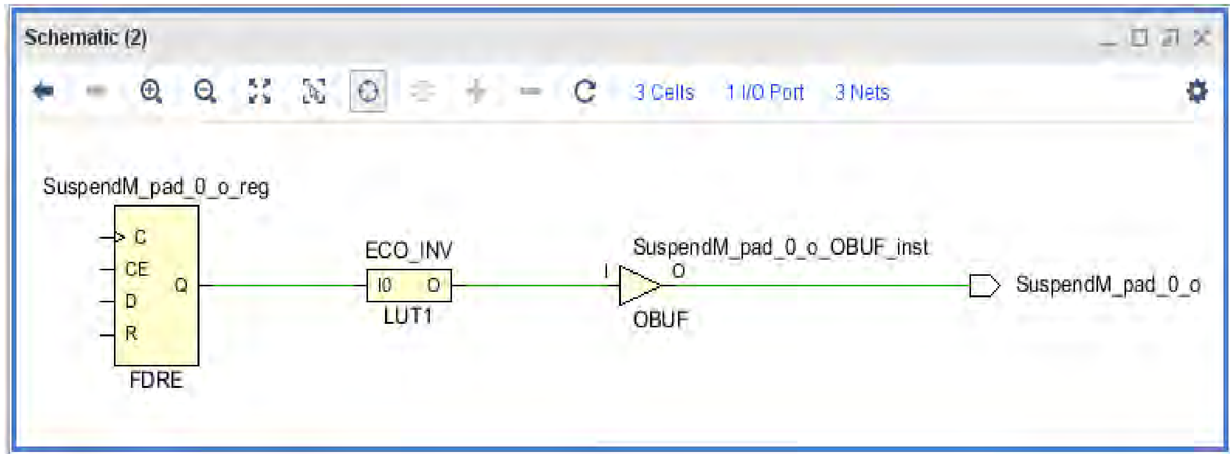


Figure 3-29: **Schematic Showing Netlist Changes After Adding Inverter**

After the netlist has been successfully modified, the logical changes must be implemented. The `LUT1` cell must be placed, and the nets to and from the cell routed. This must occur without modifying placement or routing of parts of the design that have not been modified. The Vivado implementation commands automatically use incremental mode when `place_design` is run on the modified netlist, and the log file reflects that by showing the Incremental Placement Summary:

```
+------------------------------------------------------------------------------+
|Incremental Placement Summary                                                 |
+------------------------------------------------------------------------------+
|                          Type                         | Count  | Percentage |
+------------------------------------------------------------------------------+
|  Total instances                                     |   3834 |     100.00 |
|  Reused instances                                    |   3833 |      99.97 |
|  Non-reused instances                                |      1 |       0.03 |
|    New                                               |      1 |       0.03 |
+------------------------------------------------------------------------------+
```

To preserve existing routing and route only the modified nets, use the `route_design` command. This incrementally routes only the changes, as you can see in the Incremental Routing Reuse Summary in the log file:

```
-------------------------------------------------------
|Incremental Routing Reuse Summary                     |
-------------------------------------------------------
|Type                     | Count      | Percentage    |
-------------------------------------------------------
|Fully reused nets        |        6401|          99.97 |
|Partially reused nets    |           0|           0.00 |
|Non-reused nets          |           2|           0.03 |
-------------------------------------------------------
```

Instead of automatically placing and routing the modified netlist using the incremental `place_design` and `route_design` commands, the logical changes can be committed using manual placement and routing constraints. For more information see the Modifying Placement and Modifying Routing sections earlier in this chapter.

Use Case 2: Adding a Debug Port

You can easily route an internal signal to a debug port with a netlist change. The schematic below shows the pin `demuxState_reg/Q`, which you can observe on an external port of the device.

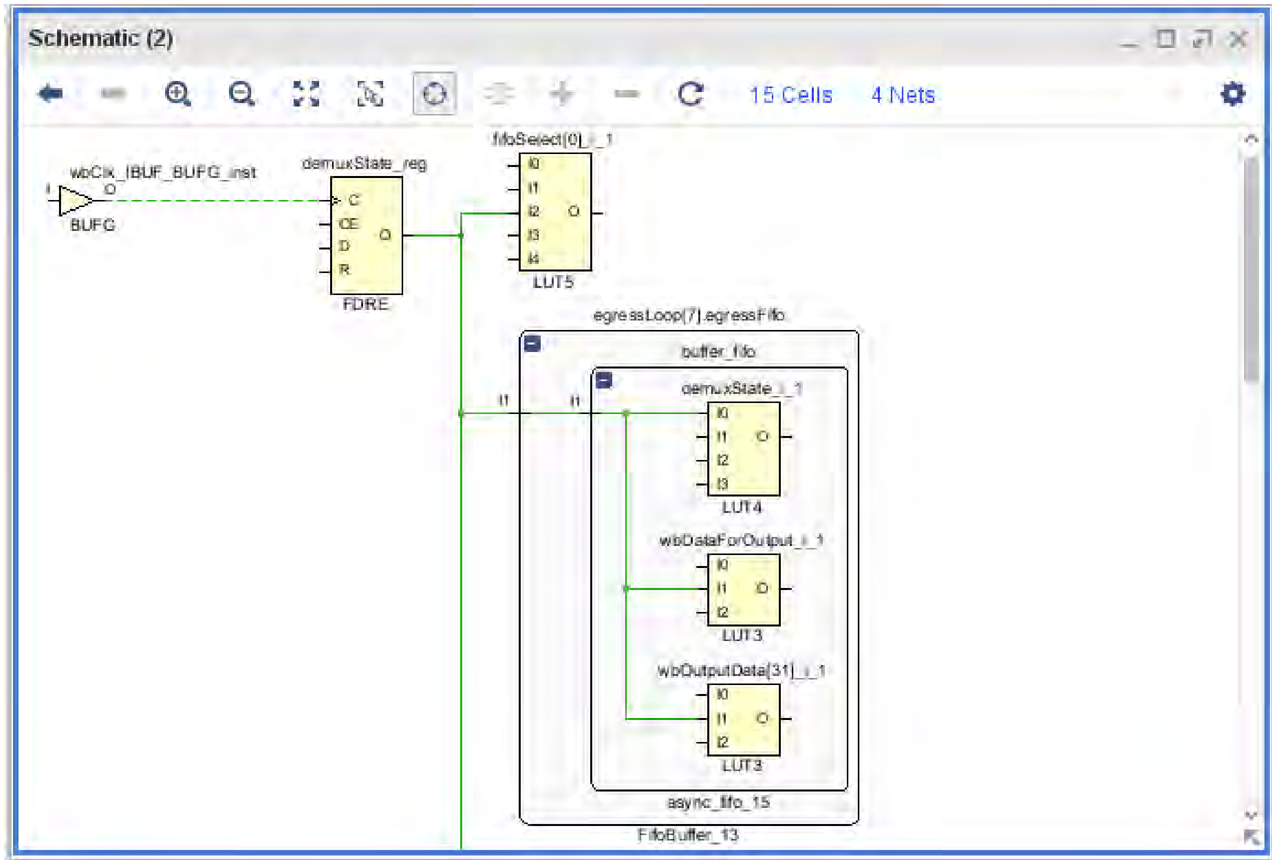


Figure 3-30: **Schematic Showing demuxState_reg**

The following Tcl script shows how to add a port to the existing design and route the internal signal to the newly created port.

```
create_port -direction out debug_port_out
set_property PACKAGE_PIN AB20 [get_ports {debug_port_out}]
set_property IOSTANDARD LVCMOS18 [get_ports [list debug_port_out]]
create_cell -reference [get_lib_cells [get_libs]/OBUF] ECO_OBUF1
create_net ECO_OBUF1_out
connect_net -net ECO_OBUF1_out -objects ECO_OBUF1/O
connect_net -net ECO_OBUF1_out -objects [get_ports debug_port_out]
connect_net -net [get_nets -of [get_pins demuxState_reg/Q]] -objects ECO_OBUF1/I
```

The example script accomplishes the following:

- Creates a debug port.
    - Assigns it to package pin AB20.
    - Assigns it an I/O standard of LVCMOS18.
- Creates an OBUF that drives the debug port through net ECO_OBUF1_out.
- Creates a net to connect the output of the demuxState_reg register to the input of the OBUF.

Figure 3-31 shows the schematic of the resulting logical netlist changes.

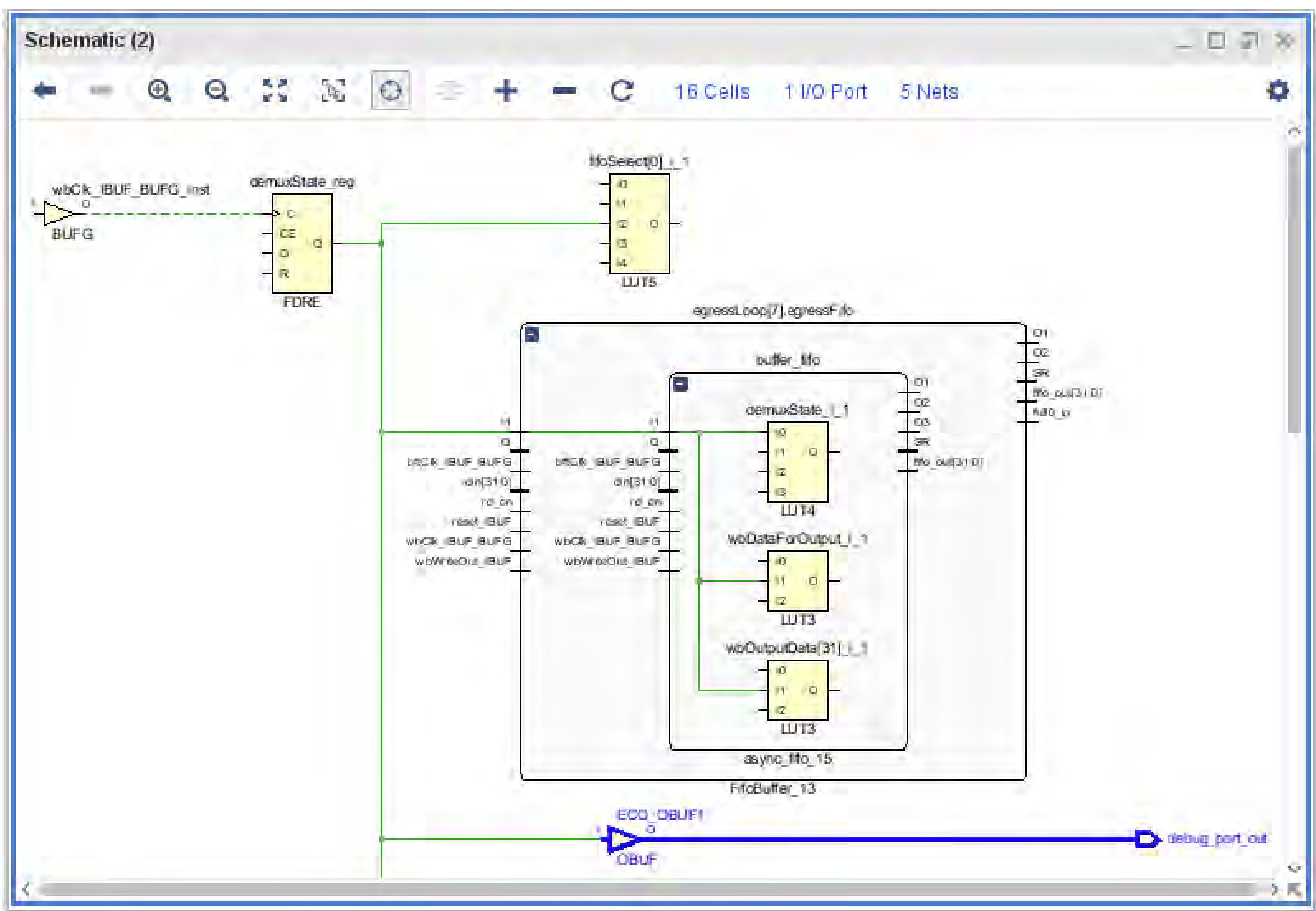


Figure 3-31: **Schematic after Adding/Routing a Debug Port**

After the netlist has been successfully modified, the logical changes must be implemented. Because the port has been assigned to a package pin, the OBUF driving the port is automatically placed in the correct location. Therefore, the placer does not have anything to place and therefore incremental compile is not triggered when running `place_design` followed by `route_design`. To route the newly added net that connects the internal signal to the `OBUF` input, use the `route_design -nets` command or route the net manually to avoid a full `route_design` pass which might change the routing for other nets. Alternatively, you can run `route_design -preserve`, which preserves existing routing. See Using Other route_design Options, page 104.

Use Case 3: Adding a Pipeline Stage to Improve Timing

Adding registers along a path to split combinational logic into multiple cycles is called *pipelining*. Pipelining improves register-to-register performance by introducing additional latency in the pipelined path. Whether pipelining works depends on the latency tolerance of your design. The schematic in Figure 3-32 shows the critical path originating at a `RAMB36E1` and going through two `LUT6` cells before terminating at an `FF`. Adding a pipeline stage can improve timing for the critical path and can be accomplished by modifying the netlist.

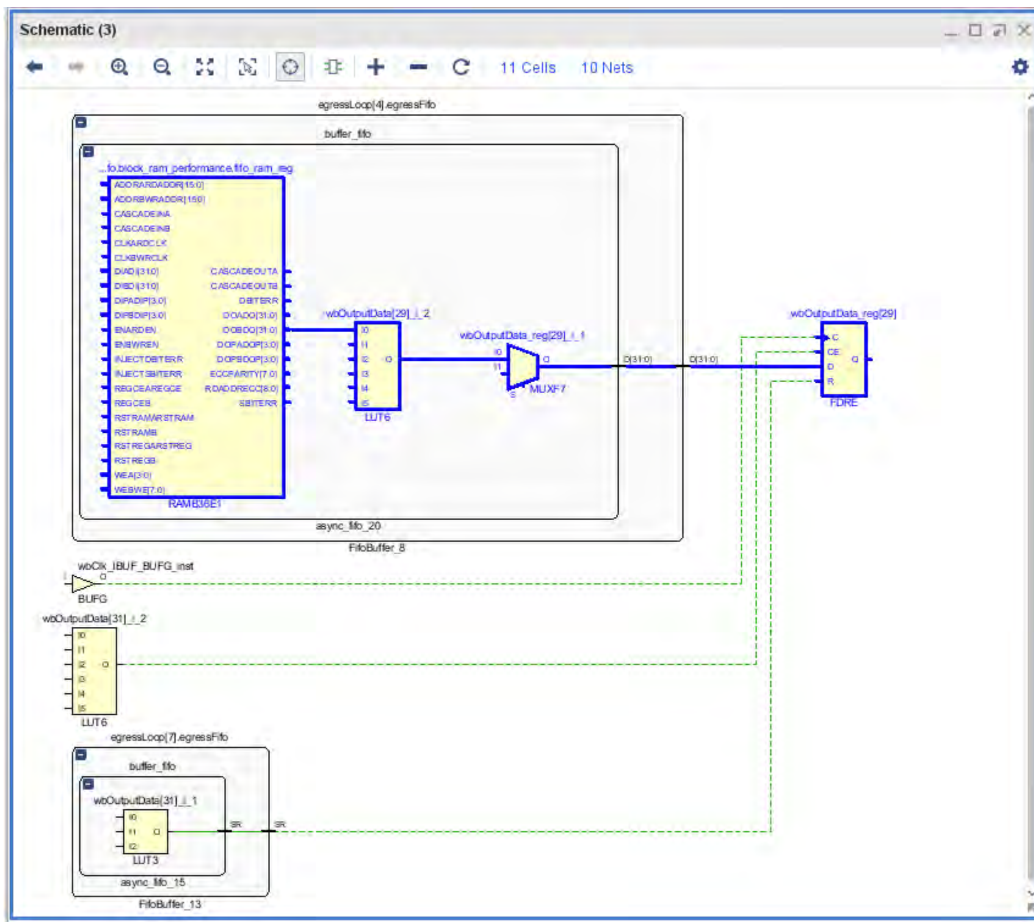


Figure 3-32: **Schematic Prior to Addition of Pipeline Register**

The following Tcl script shows how to insert a pipeline register between the two `LUT6` cells. The register is implemented with the same control signals as the load register.

```
create_cell -reference [get_lib_cells -of [get_cells {wbOutputData_reg[29]}]]
ECO_pipe_stage[29]
foreach control_pin {C CE R} {
  connect_net -net [get_nets -of [get_pins wbOutputData_reg[29]/${control_pin}]] \
    -objects [get_pins ECO_pipe_stage[29]/${control_pin}]
}
disconnect_net -objects \
{egressLoop[4].egressFifo/buffer_fifo/infer_fifo.block_ram_performance.fifo_ram_reg/DOBDO[
29]}
create_net {egressLoop[4].egressFifo/buffer_fifo/ECO_pipe_stage[29]_in}
connect_net -hierarchical -net
{egressLoop[4].egressFifo/buffer_fifo/ECO_pipe_stage[29]_in} -objects \ [list \
{ECO_pipe_stage[29]/D} \

{egressLoop[4].egressFifo/buffer_fifo/infer_fifo.block_ram_performance.fifo_ram_reg/DOBDO[
29]}]
connect_net -hierarchical -net {egressLoop[4].egressFifo/buffer_fifo/dout2_in[29]}
-objects [list \ {ECO_pipe_stage[29]/Q}]
```

The picture below shows the schematic of the resulting logical netlist changes.

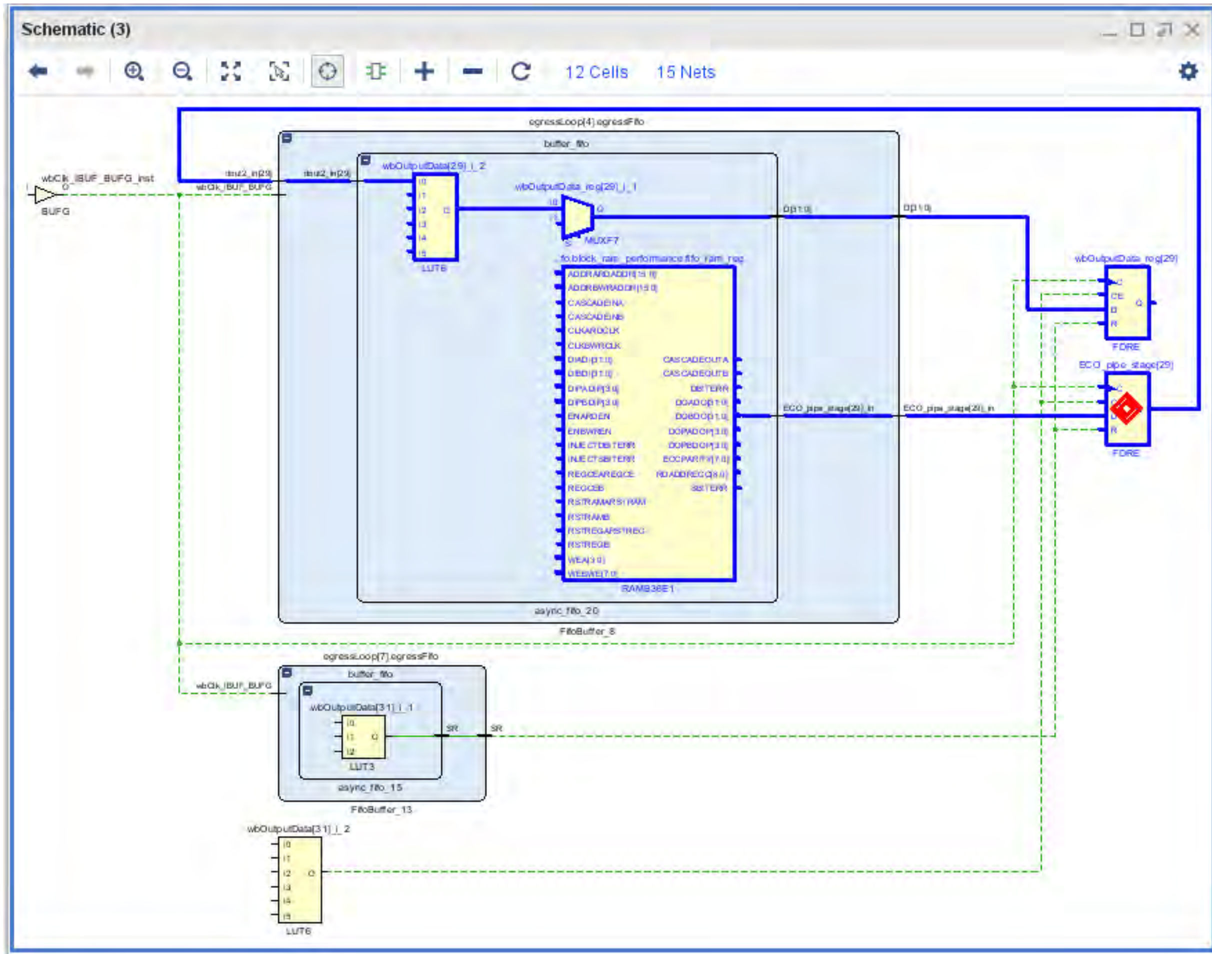


Figure 3-33: **Schematic Showing Addition of Pipeline Register**

After the netlist has been successfully modified, the logical changes must be committed. Accomplish this using the `place_design` and `route_design` commands.

Vivado ECO Flow

IMPORTANT: *ECOs only work on design checkpoints. The ECO Layout is only available after a design checkpoint has been opened in the Vivado IDE.*

Engineering change orders (ECOs) are modifications to the post implementation netlist with the intent to implement the changes with minimal impact to the original design. Vivado provides an ECO flow, which allows you to modify a design checkpoint, implement the changes, run reports on the changed netlist, and generate programming files.

Common use cases for the ECO flow are:

- Modifying debug probes of ILA and/or VIO cores in the design
- Routing an internal net to a package pin for external probing
- Evaluating what-if scenarios (improving timing, fixing logic bugs, and so on)

The advantage of the ECO flow is fast turn-around time by taking advantage of the incremental place and route features of the Vivado tool.

The Vivado IDE provides a predefined layout to support the ECO flow.

To access the ECO Layout, select **Layout > ECO**.

ECO Navigator

The ECO Navigator provides access to the commands that are required to complete an ECO.

Scratch Pad

The scratch pad tracks netlist changes and place and route status for Cells, Pins, Ports, and Nets.

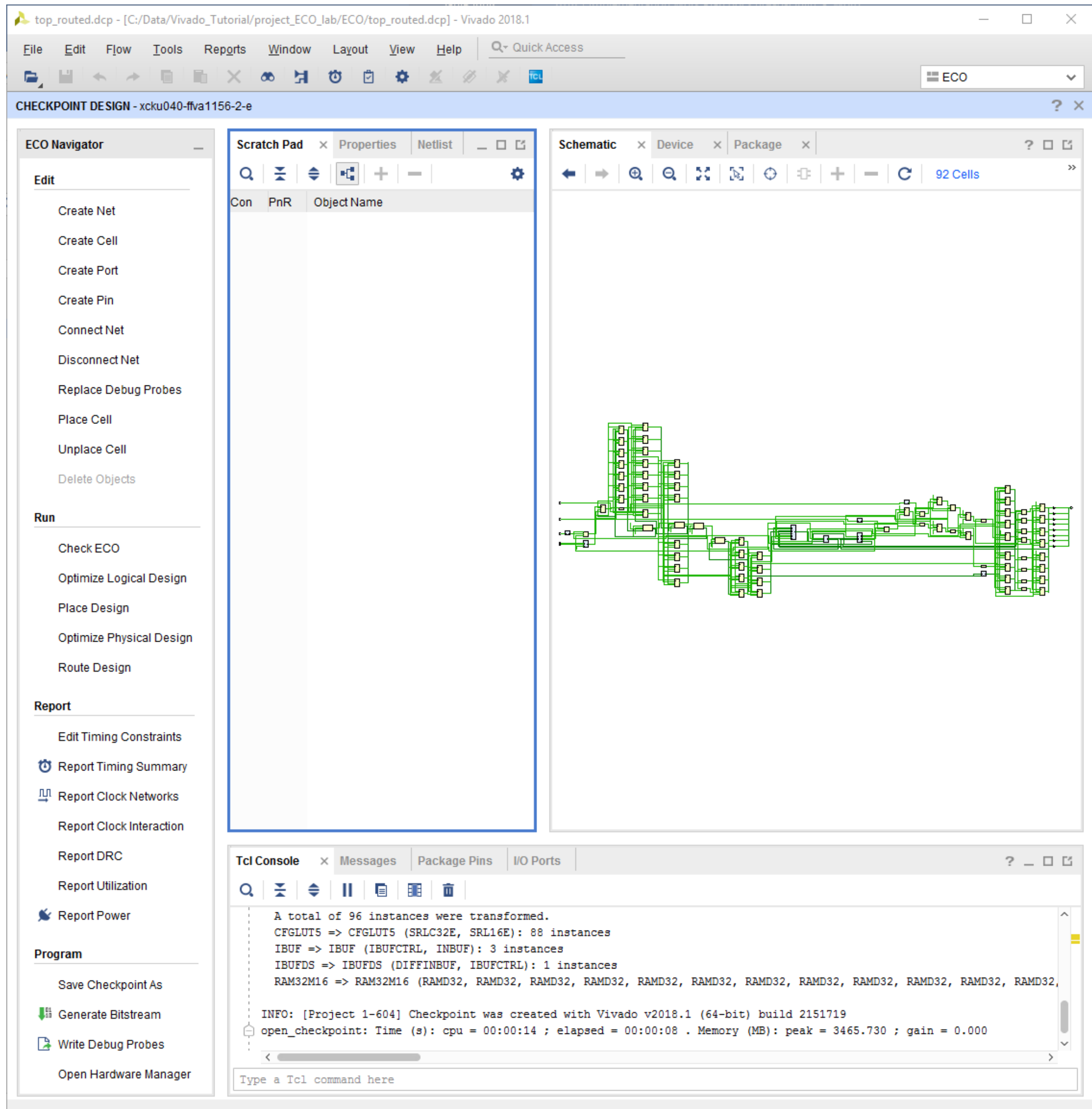


Figure 3-34: **Vivado ECO Layout**

ECO Flow Chart

The figure below shows a typical ECO flow. You open a previously implemented design. After modifying the netlist, if the design is not fully placed, you run Incremental Place. Otherwise you can skip straight to Incremental route. After that you can save your changes to a new checkpoint and write new programming and debug probe files and Open the Hardware manger to program your device. If you are satisfied with your changes you can incorporate them into your original design. Otherwise, you can start at the beginning of the ECO flow until the design is working as expected.

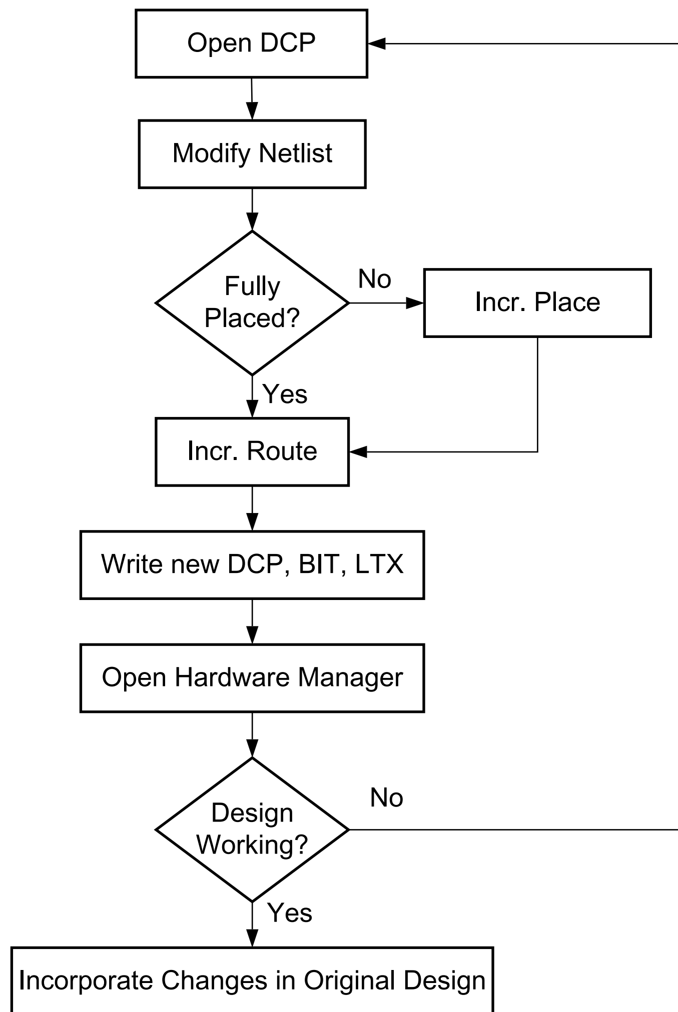


Figure 3-35: **ECO Flow Chart**

TIP: *When you re-run implementation in project mode the results in the previous run directory will be deleted. Save the ECO checkpoint to a new directory or create a new implementation run for your subsequent compile to preserve the changes to the ECO checkpoint.*

ECO Navigator Use

The ECO Navigator provides access to all of the commands required to complete an ECO. The ECO Navigator is divided into four sections: Edit, Run, Report, and Program.

Edit Section

The Edit section of the ECO Navigator (shown in the below figure) provides access to all the commands that are required to modify the netlist.

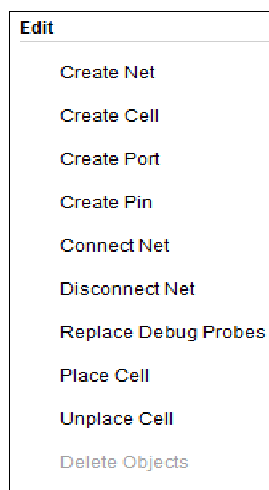


Figure 3-36: **ECO Navigator Edit Commands**

Create Net: Opens the Create Net dialog box, which allows you to create new nets in the current loaded design. Nets can be created hierarchically from the top level of the design, or within any level of the hierarchy by specifying the hierarchical net name. Bus nets can be created with increasing or decreasing bus indexes, using negative and positive index values. To create a bus net, turn on Create bus and specify the beginning and ending index values. If you select a pin or port, you can have the newly created net automatically connect to them by selecting the **Connect selected pins and ports** check box.

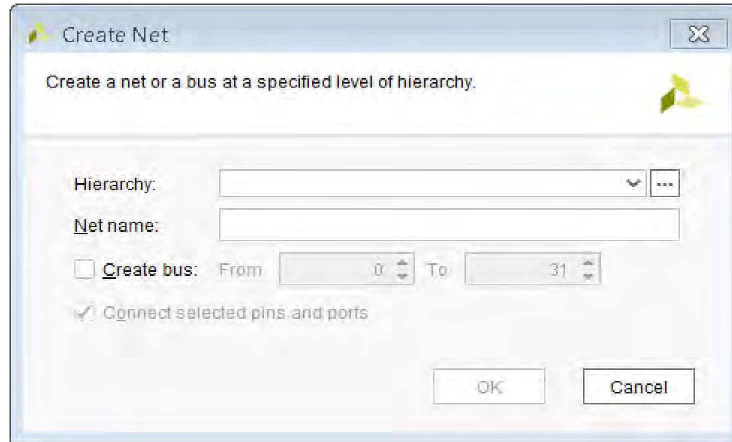


Figure 3-37: **Create Net Dialog Box**

Create Cell: Opens the Create Cell dialog box, which allows you to add cells to the netlist of the currently loaded design. You can add new cell instances to the top- level of the design, or hierarchically within any module of the design. Instances can reference an existing cell from the library or design source files, or you can add a black box instance that references cells that have not yet been created. If a LUT cell is created, you can specify a LUT equation in the Specify LUT Equation dialog box by selecting it.

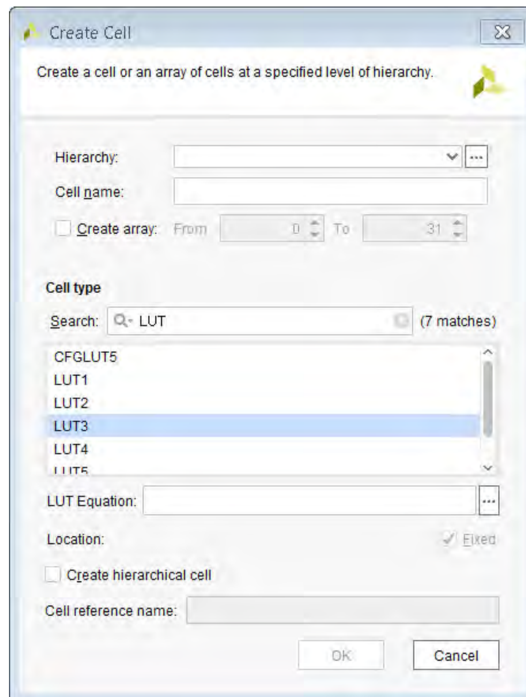


Figure 3-38: **Create Cell Dialog Box**

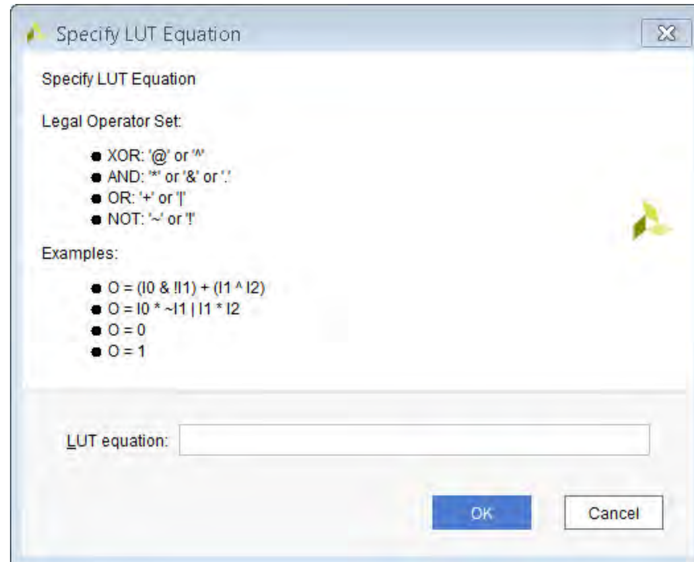


Figure 3-39: **Specify LUT Equation Dialog Box**

Create Port: Opens the Create Port dialog box, in which you can create a port and specify such parameters as direction, width, single-ended, or differential. New ports are added at the top level of the design hierarchy. You can create bus ports with increasing or decreasing bus indexes, using negative and positive index values. You can also specify I/O standard, pull type, and ODT type. When a Location is specified, the port is assigned to a package pin.

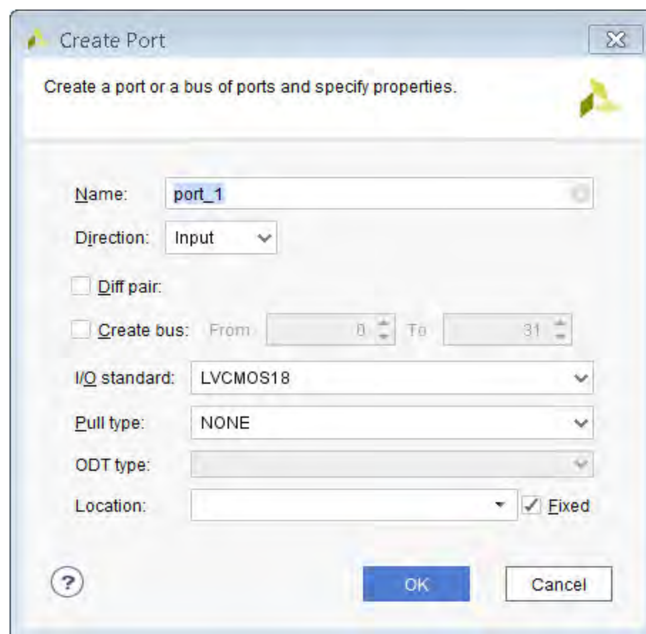


Figure 3-40: **Create Port Dialog Box**

Create Pin: Opens the Create Pin dialog box, which allows you to add single pins or bus pins to the current design. You can define attributes of the pin, such as direction and bus width, as well as the pin name. You can create bus pins with increasing or decreasing bus indexes, using negative and positive index values. A pin must be created on an existing cell instance, or it is considered a top-level pin, which should be created using the `create_port` command. If the instance name of a cell is not specified, the pin cannot be created.

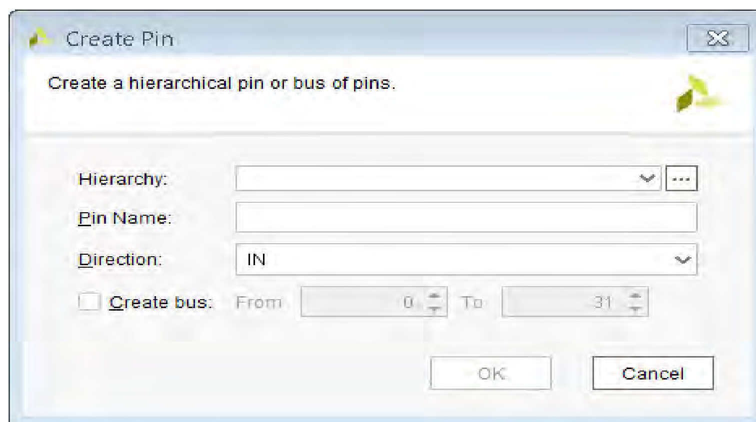


Figure 3-41: **Create Pin Dialog Box**

Connect Net: The selected pin or port is connected to the selected net. If a net is not selected, the Connect Net dialog box opens, which allows you to specify a net to connect to the selected pins or ports in the design. The window displays a list of nets at the current selected level of hierarchy that can be filtered dynamically by typing a net name in the search box. The selected net will be connected across levels of hierarchy in the design, by adding pins and hierarchical nets as needed to complete the connection.

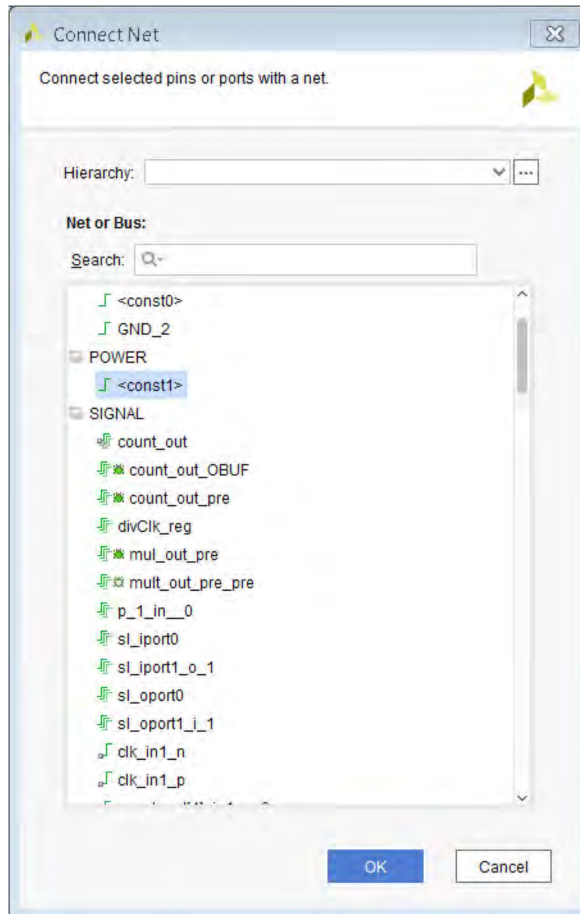


Figure 3-42: **Connect Net Dialog Box**

Disconnect Net: Disconnects the selected net, pin, port or cell from the net in the current design. If a cell is selected, all nets connected to that cell will be disconnected.

Replace Debug Probes: Opens the Replace Debug Probes dialog box, if a Debug core has previously been inserted into the design. The Replace Debug Probes dialog box contains information about the nets that are probed in your design using the ILA and/or VIO cores. You can modify the nets that are connected to the debug probe by clicking the icon next to the net name in the Probe column. This opens the Choose Nets dialog box, which allows you to select a new net to connect to the debug probe.

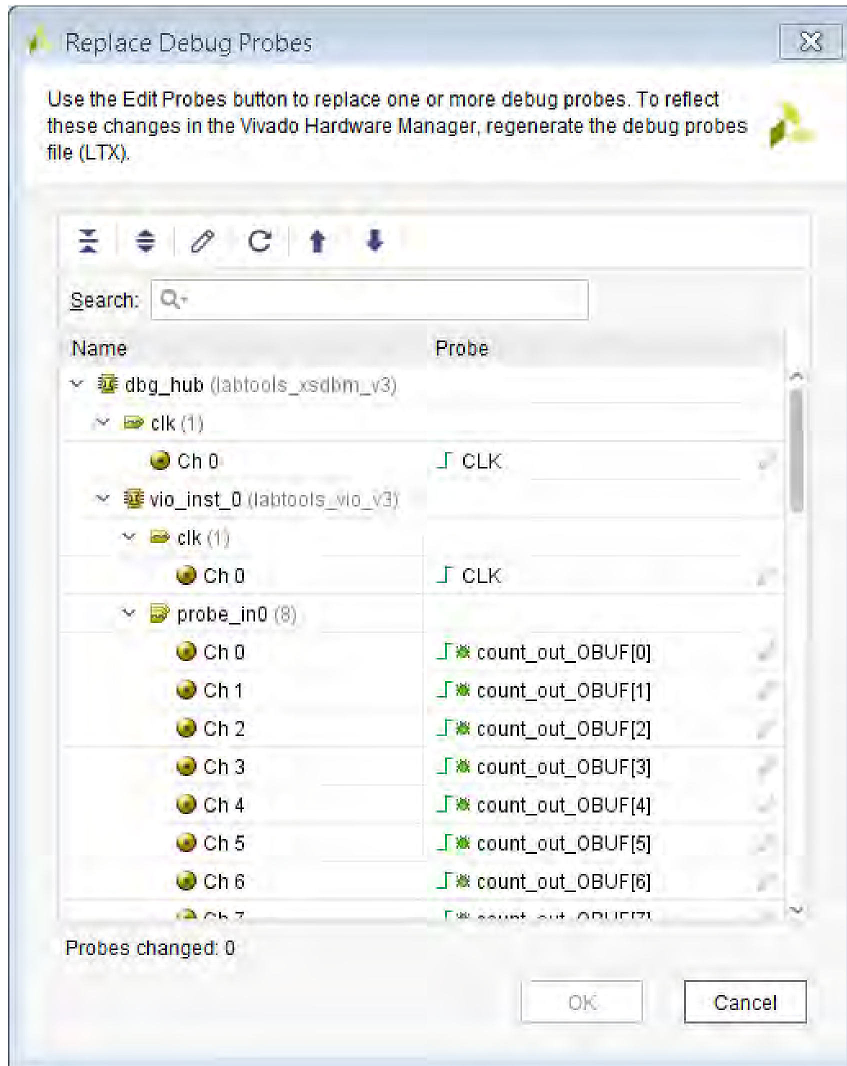


Figure 3-43: **Replace Debug Probes Dialog Box**

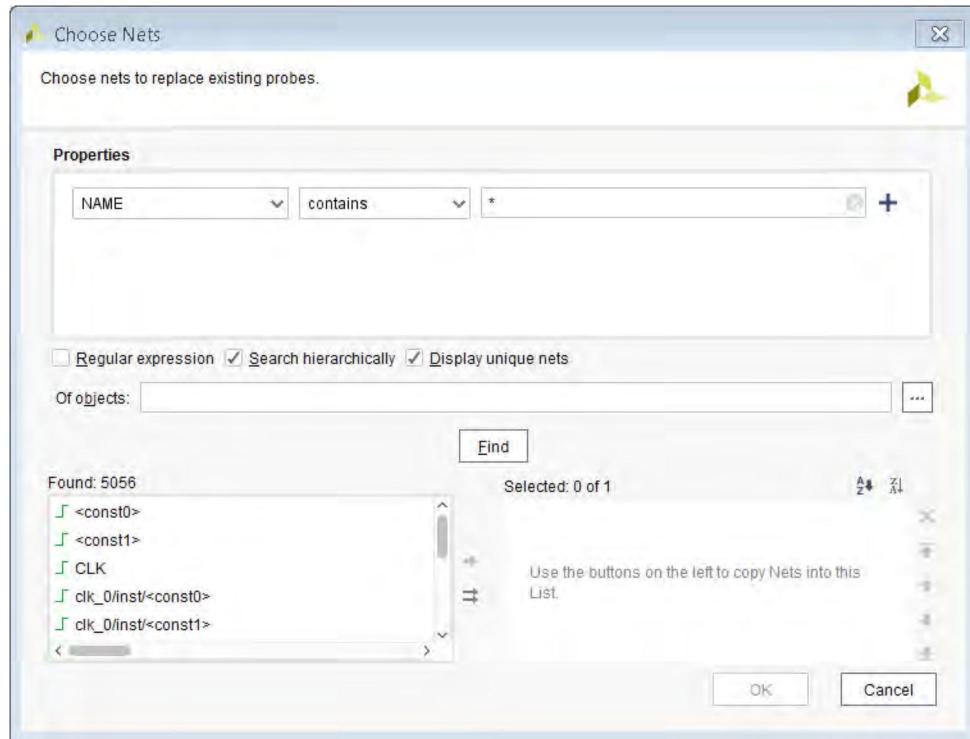


Figure 3-44: **Choose Nets Dialog Box**

Place Cell: Places the selected cell onto the selected device resource.

Unplace Cell: Unplaces the selected cell from its current placement site.

Delete Objects: Deletes the selected objects from the current design.

Run Section

The Run Section of the ECO Navigator, shown in the figure below, provides access to all the commands required to implement the current changes.

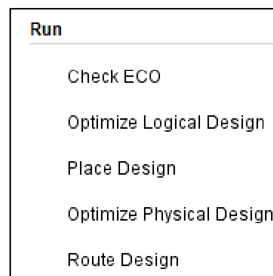


Figure 3-45: **ECO Navigator Run Commands**

Check ECO: Runs the ECO checks rule deck on the current design.

TIP: *The Vivado tools allows you to make netlist changes unconditionally using the ECO commands. However, logical changes can lead to invalid physical implementation. Run the* ***Check ECO*** *function to flag any invalid netlist changes or new physical restrictions that need to be addressed before physical implementation can commence.*

Optimize Logical Design: In some cases, it is desirable to run `opt_design` on the modified design to optimize the netlist. This command opens the Optimize Logical Design dialog box, allowing you to specify options for the `opt_design` command. Any options that are entered in the dialog box are appended to the `opt_design` command as they are typed. For example, to run `opt_design -sweep`, type `-sweep` under **Options**.

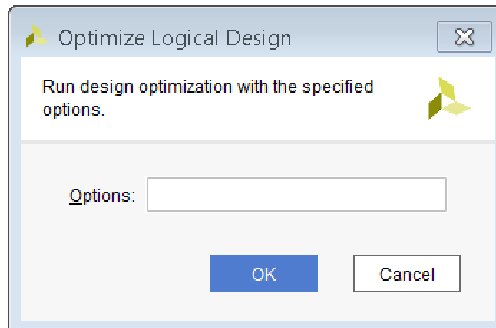


Figure 3-46: **Optimize Logical Design Dialog Box**

Place Design: Runs incremental `place_design` on the modified netlist as long as 75% or more of the placement can be reused. The Incremental Placement Summary at the end of `place_design` provides statistics on incremental reuse. Selecting this command opens the Place Design dialog box and allows you to specify options for the `place_design` command. Any options that are entered in the dialog box are appended to the `place_design` command as they are typed.

Refer to Incremental Implementation, page 110 for additional information on Incremental Place and Route.

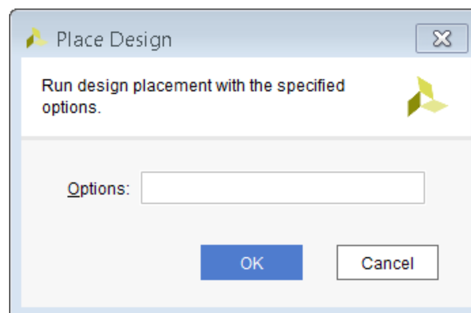


Figure 3-47: **Place Design Dialog Box**

Optimize Physical Design: In some cases it is desirable to run `phys_opt_design` on the modified design to perform physical optimization on the netlist. This command opens the Optimize Physical Design dialog box and allows you to specify options for the `phys_opt_design` command. Any options that are entered in the dialog box are appended to the `phys_opt_design` command as they are typed. For example, to run `phys_opt_design -fanout_opt`, type `-fanout_opt` under **Options**.

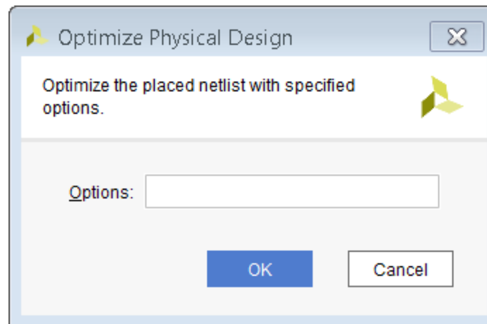


Figure 3-48: **Optimize Physical Design Dialog Box**

Route Design: Selecting this command opens the Route Design dialog box. Depending on the selection, this command allows you to perform an Incremental Route of the modifications made to the design, Route the selected pin, or Route selected nets. If Incremental Route is selected on a modified netlist that has less than 75% of reused nets, the tool reverts to the non-incremental `route_design`.

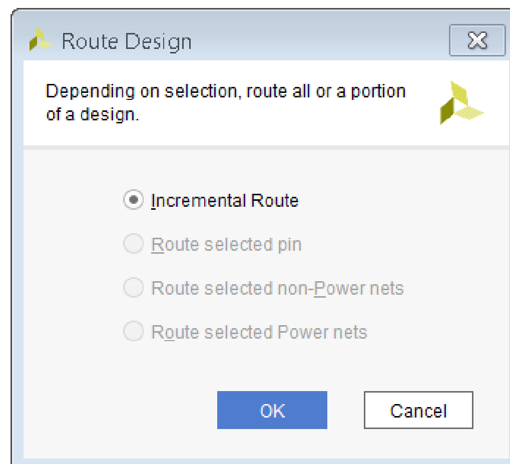


Figure 3-49: **Route Design Dialog Box**

Refer to Incremental Implementation, page 110 for additional information on incremental Place and Route.

Depending on your selection, you have four options to route the ECO changes:

- Incremental Route: This is the default option.
- Route selected pin: This option limits the route operation to the selected pin.
- Route selected non-Power nets: This option routes only the selected signal nets.
- Route selected Power nets: This option routes only the selected VCC/GND nets.

Report Section

The Report Section of the ECO Navigator, shown in the figure below, provides access to all the commands that are required to run reports on the modified design.

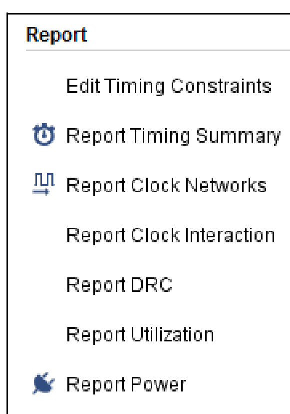


Figure 3-50: **ECO Navigator Report Commands**

For more information on these commands, refer to *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893) [Ref 3].

Program Section

The Program Section of the ECO Navigator, shown in the figure below, provides access to the commands that allow you to save your modifications, generate a new BIT file for programming and a new LTX file for your debug probes, and program the device.

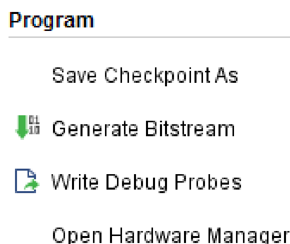


Figure 3-51: **ECO Navigator Program Commands**

Save Checkpoint As: This command allows you to save your modifications to a new checkpoint.

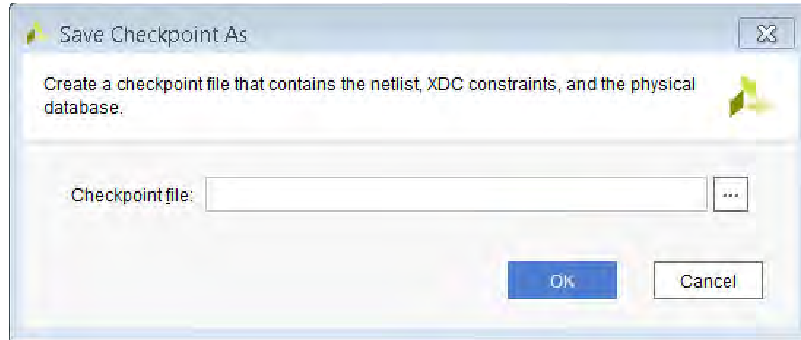


Figure 3-52: **Save Checkpoint As Dialog Box**

Generate Bitstream: This command allows you to generate a new `.bit` file for programming.

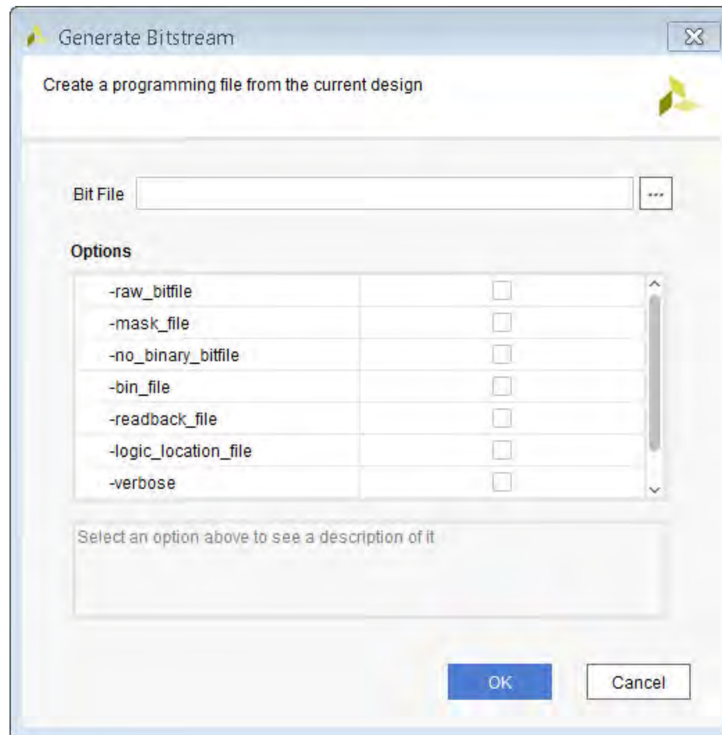


Figure 3-53: **Generate Bitstream Dialog Box**

Write Debug Probes: This command allows you to generate a new .ltx file for your debug probes. If you made changes to your debug probes using the Replace Debug Probes command, you need to save the updated information to a new debug probes file (LTX) to reflect the changes in the Vivado Hardware Manager.

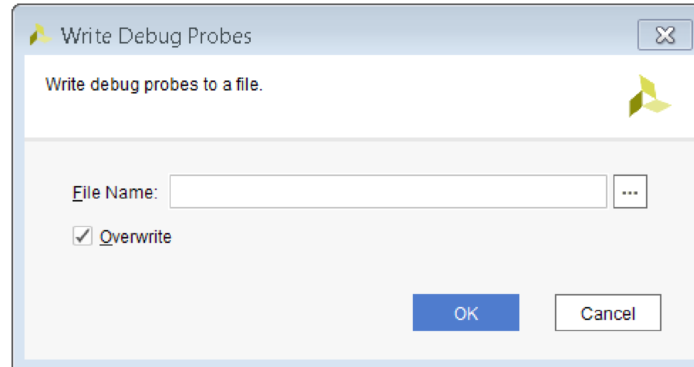


Figure 3-54: **Write Debug Probes Dialog Box**

Scratch Pad

The Scratch Pad is updated as changes are made to the loaded checkpoint. See the following figure. The **Object Name** column displays hierarchical names of Cells, Nets, Ports, and Pins. The Connectivity (**Con**) column tracks the connectivity of the objects and the Place and Route (**PnR**) column tracks the place and route status of the objects. In the Scratch Pad shown in the following figure, notice that check marks in the **Con** and **PnR** columns identify connectivity and place/route status. Looking at this figure, you can identify the following:

- The port `ingressFifoWrEn_debug` has been added and assigned to a package pin.
- The net `ingressFifoWrEn` has been connected to the newly created Port, but the connection has not yet been routed to the port.

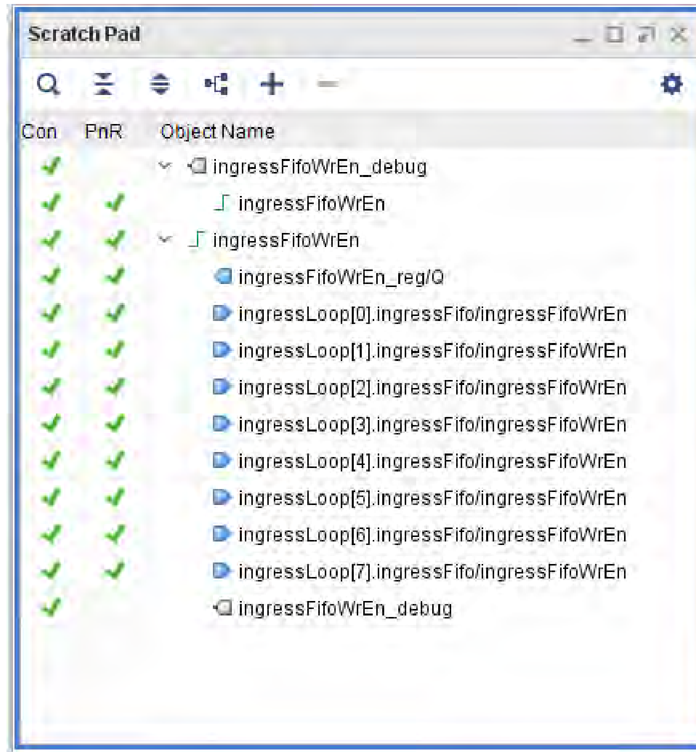


Figure 3-55: **Scratch Pad**

Scratch Pad Toolbar Commands

The Scratch Pad commands are:

- **Search**: Searches the Scratch Pad for objects by name.
- **Collapse All**: Displays objects by groups, and does not display individual members of the group.
- **Expand All**: Shows an expanded view of all members of a group.
- **Group by Type**: Displays the objects by type, or in the order they have been added.
- **Add selected objects**: Adds selected objects to the Scratch Pad.
- **Remove selected objects**: Removes selected objects from the Scratch Pad.

Scratch Pad Pop-up Menu

When you right-click in the Scratch Pad, the following pop-up menu commands are available:

- **Clear Scratch Pad**: Clears the contents of the Scratch Pad.
- **Add Objects to Scratch Pad**: Adds unconnected, unplaced, or unrouted objects to the Scratch Pad.

- **Select Array Elements**: Selects all the elements in an array if one element has been selected.
- **Clone**: Creates a copy of the selected object.
- **Connect Net to Output Port**: Opens the Connect Net to Output Port dialog box, which allows you to connect the selected net to an external port. See Figure 3-56.
- **Elide Setting**: Specifies how to truncate long object names that don't fit in the Object Name column. Choices are Left, Middle, and Right.
- **Object Properties**: Opens the Object Properties dialog box.
- **Report Net Route Status**: Reports the route status of the selected net.
- **Select Driver Pin**: Selects the driver pin of the selected net.
- **Unplace**: Unplaces the selected I/O ports.
- **Configure I/O Ports**: Assigns various properties of the selected I/O ports.
- **Split Diff Pair**: Removes the differential pair association from the selected port.
- **Auto-place I/O Ports**: Places I/O ports using the Autoplace I/O Ports wizard.
- **Place I/O Ports in Area**: Assigns the currently selected ports onto pins in the specified area.
- **Place I/O Ports Sequentially**: Assigns the currently selected ports individually onto package pins.
- **Fix Ports**: Fixes the selected placed I/O ports.
- **Unfix Ports**: Unfixes the selected placed I/O ports.
- **Floorplanning**: Assign selected cells to Pblock.
- **Highlight Leaf Cells**: Highlights the primitive logic for the selected cell.
- **Unhighlight Leaf Cells**: Unhighlights the primitive logic for the selected cell.
- **Delete**: Deletes the selected objects.
- **Highlight**: Highlights the selected objects.
- **Unhighlight**: Unhighlights the selected objects.
- **Mark**: Draws a marker for the selected object.
- **Unmark**: Removes the marker for the selected object.
- **Schematic**: Creates a schematic from the selected objects.
- **Show Connectivity**: Shows the connectivity of the selected object.
- **Find**: Opens the Find dialog box to find objects in the current design or device by filtering TCL properties and objects.

- **Export to Spreadsheet**: Writes the contents of the Scratch Pad to a Microsoft Excel spreadsheet.

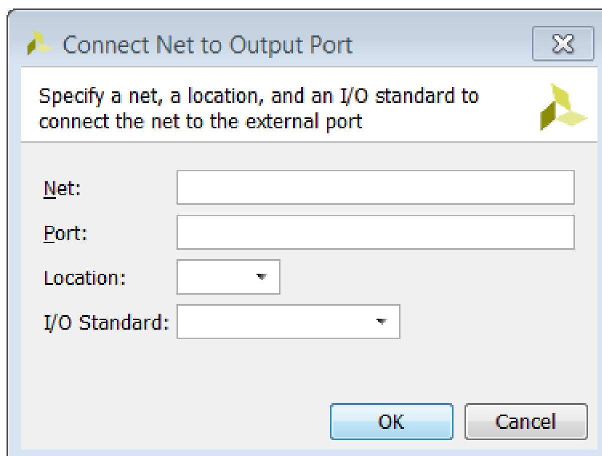


Figure 3-56: **Connect Net to Output Port Dialog Box**

Schematic Window

Logical changes are reflected in the schematic view as soon as the netlist is changed. The following figure shows an updated schematic based on the netlist changes shown in Figure 3-56.

TIP: *Use the Mark Objects and Highlights Objects command to help you keep track of objects in the Schematic Window as you make changes to the netlist.*

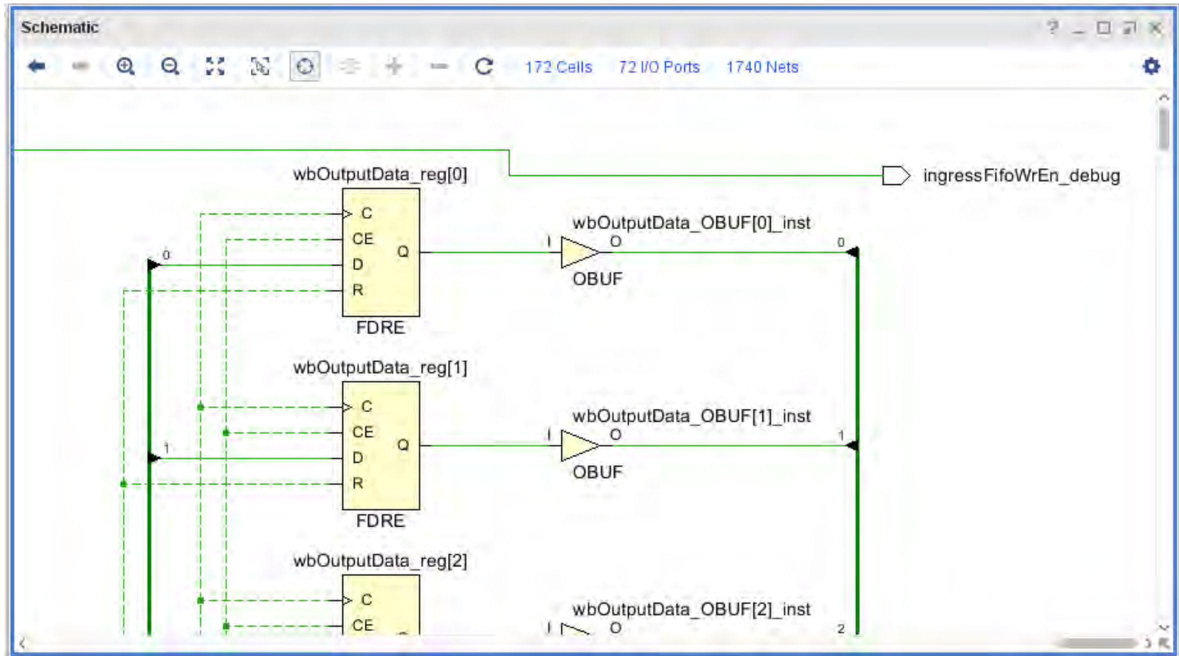


Figure 3-57: **Schematic Window**

Appendix A

Using Remote Hosts and Compute Clusters

Overview

The Xilinx® Vivado® Integrated Design Environment (IDE) supports simultaneous parallel execution of synthesis and implementation runs on multiple Linux hosts. You can accomplish this manually by configuring individual hosts or by specifying the commands to launch jobs on existing compute clusters.

Currently Linux is the only operating system Vivado supports for remote host configurations. Remote host settings are accessible through the Tools menu by selecting **Tools > Settings > Remote Hosts**.

Requirements

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

- Vivado tools 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.

  For Manual Configuration, if you do not have Vivado set up upon login (CSHRC or BASHRC), use the **Run pre-launch script** option, 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 might 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 might not be visible from remote machines.

Manual Configuration

Manual configuration of remote hosts allows you to specify individual machine names on which Vivado can execute. Vivado will open a Secure Shell (SSH) on these machines and spawn additional Vivado processes. Host names can be added by clicking the add button shown in Figure A-1. Once added, the number of jobs per host can selected and hosts can optionally be disabled. The specific command used to launch the jobs must be provided. Optionally, users can configure pre and post launch scripts and an email address if you desire to be notified once the jobs complete.

IMPORTANT: *Use caution when specifying the "launch jobs with" command. For example, removing BatchMode=yes might cause the remote process to hang because the Secure Shell incorrectly prompts for an interactive password.*

RECOMMENDED: *Test each host to ensure proper setup before submitting runs to the host.*

A "greedy," round-robin style algorithm is used to submit jobs to the remote hosts. Before launching runs on multiple Linux hosts it is important to configure SSH so that the host does not require a password each time you launch a remote run.

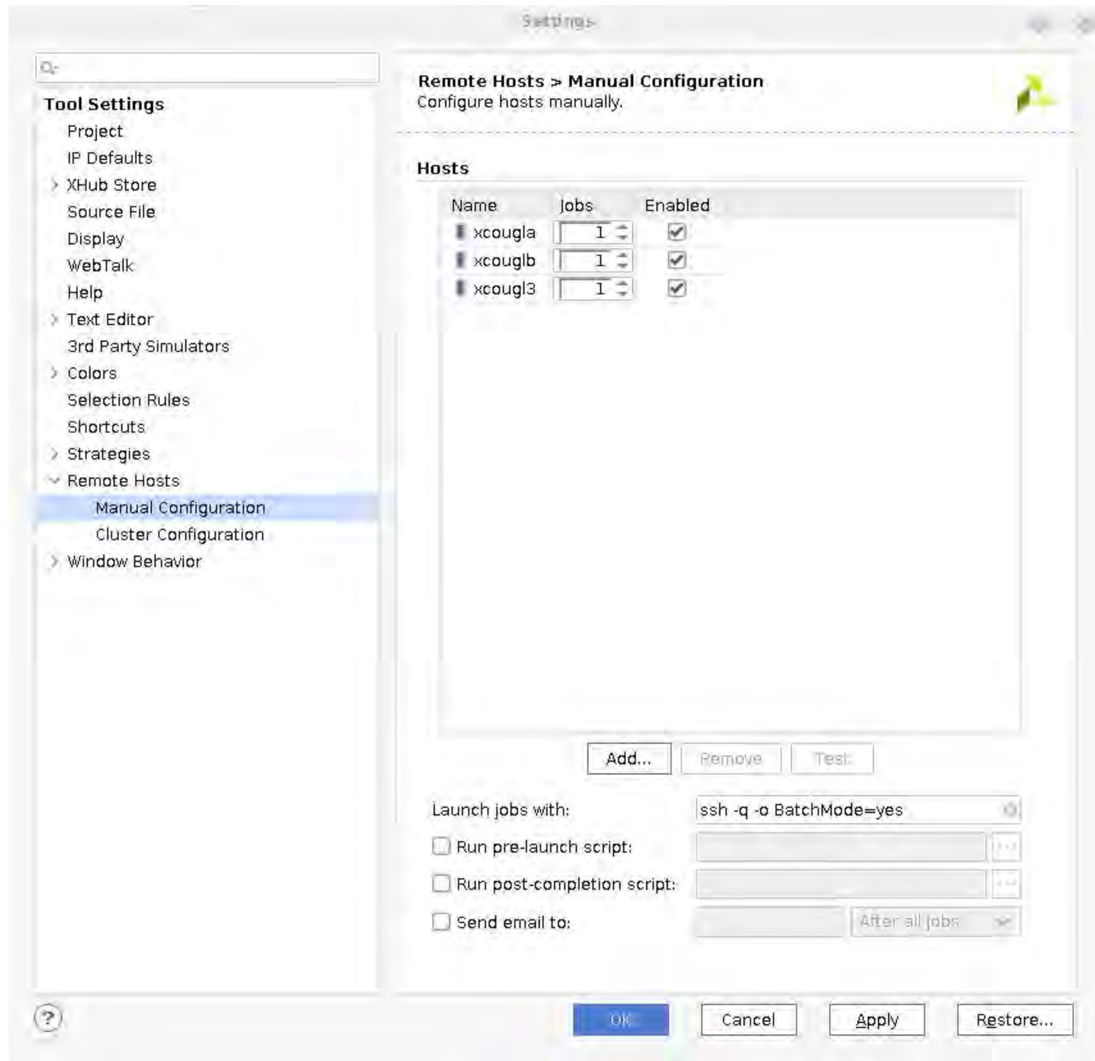


Figure A-1: **Manual Configuration of Remote Hosts**

Setting Up SSH Key Agent Forward

You can configure SSH 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 publish 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.*

Cluster Configurations

Compute Clusters are groups of machines configured through third party tools that accept jobs, schedule them, and efficiently allocate the compute resources. Common compute clusters include LSF, SGE and SLURM. To add custom compute clusters to Vivado, you can click the plus tool bar button shown in figure def and provide a name for the cluster configuration. You then need to specify the command necessary to submit a job to the cluster, cancel a job on the cluster, and the cluster type. Vivado natively support LSF, SGE and SLURM. For any other cluster you can choose CUSTOM in the combo box. The configuration can be tested by pressing the test configuration button.

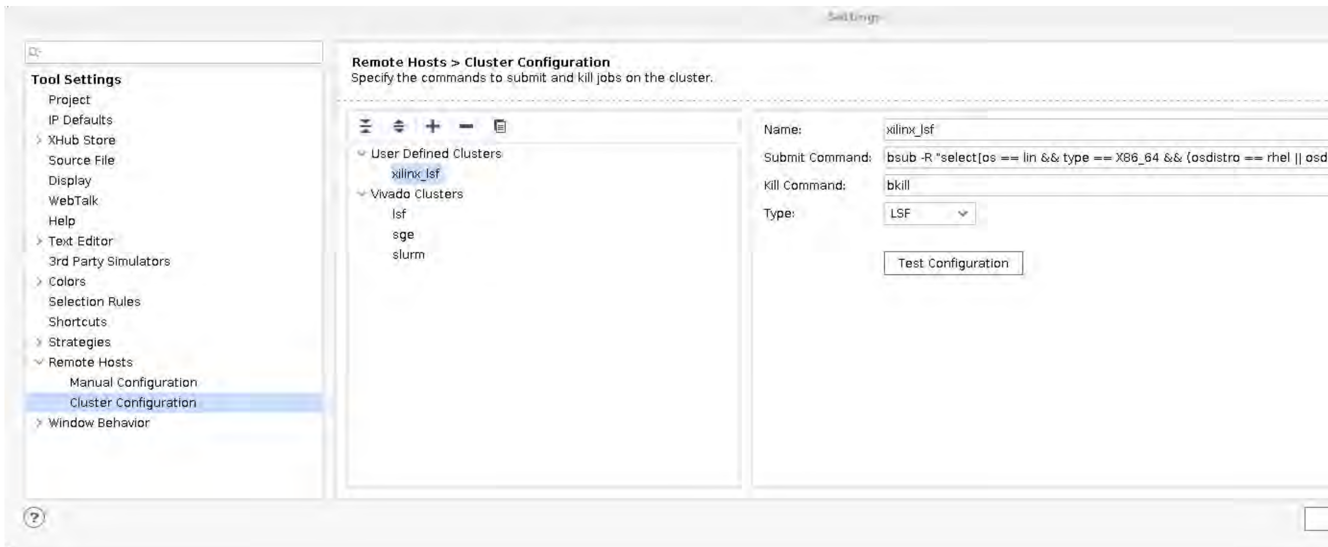


Figure A-2: **Cluster configurations Settings Dialog Box**

SLURM Specific Configuration

Configuring Vivado to run on SLURM using ssh to connect the client to the scheduler.

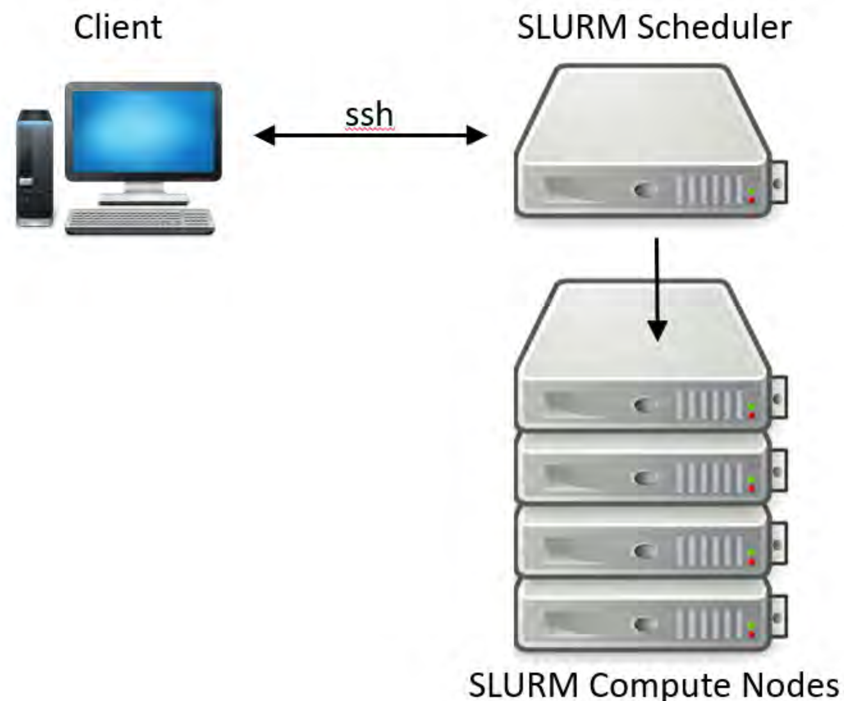


Figure A-3: SLURM Compute Nodes

In this example, the client machine name is xcolc200189, the scheduler machine name is xcolc200185.

1. Set up SSH keys on client and scheduler to enable ssh without password.
2. Start Vivado on the client machine.
3. Create a custom SLURM cluster.
   a. Open the Vivado settings window (**Tools** > **Settings**).
   b. Select Tool **Settings** > **Remote Hosts** > **Cluster Configuration**.
   c. Click the "+" button in the toolbar to create a new cluster configuration.
   d. Fill in the form as follows. Important to leave the type as **Custom**.

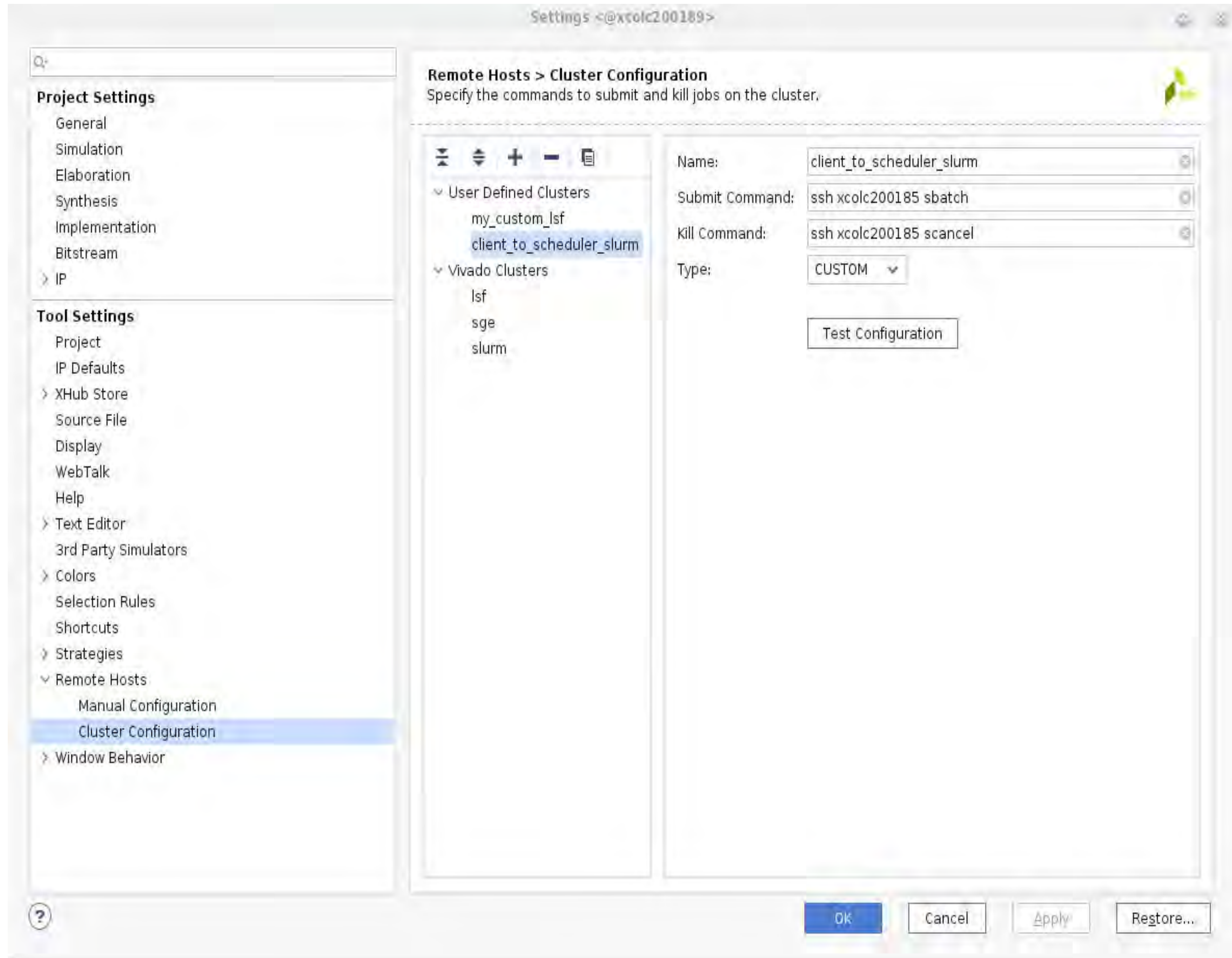


Figure A-4: Remote Hosts Cluster Configuration

4. Launch a job on the cluster to test the configuration.
   a. Select **File** > **Project** > **Open Example Project**.
   b. Next. Select **BFT** and click **Next**.
   c. Select a name and directory and click **Next**.
   d. Select the default part (xc7k70tfbg484-2) and click **Next**.
   e. Click **Finish**. In the design runs Window, select synth_1 row and click the green play toolbar button.

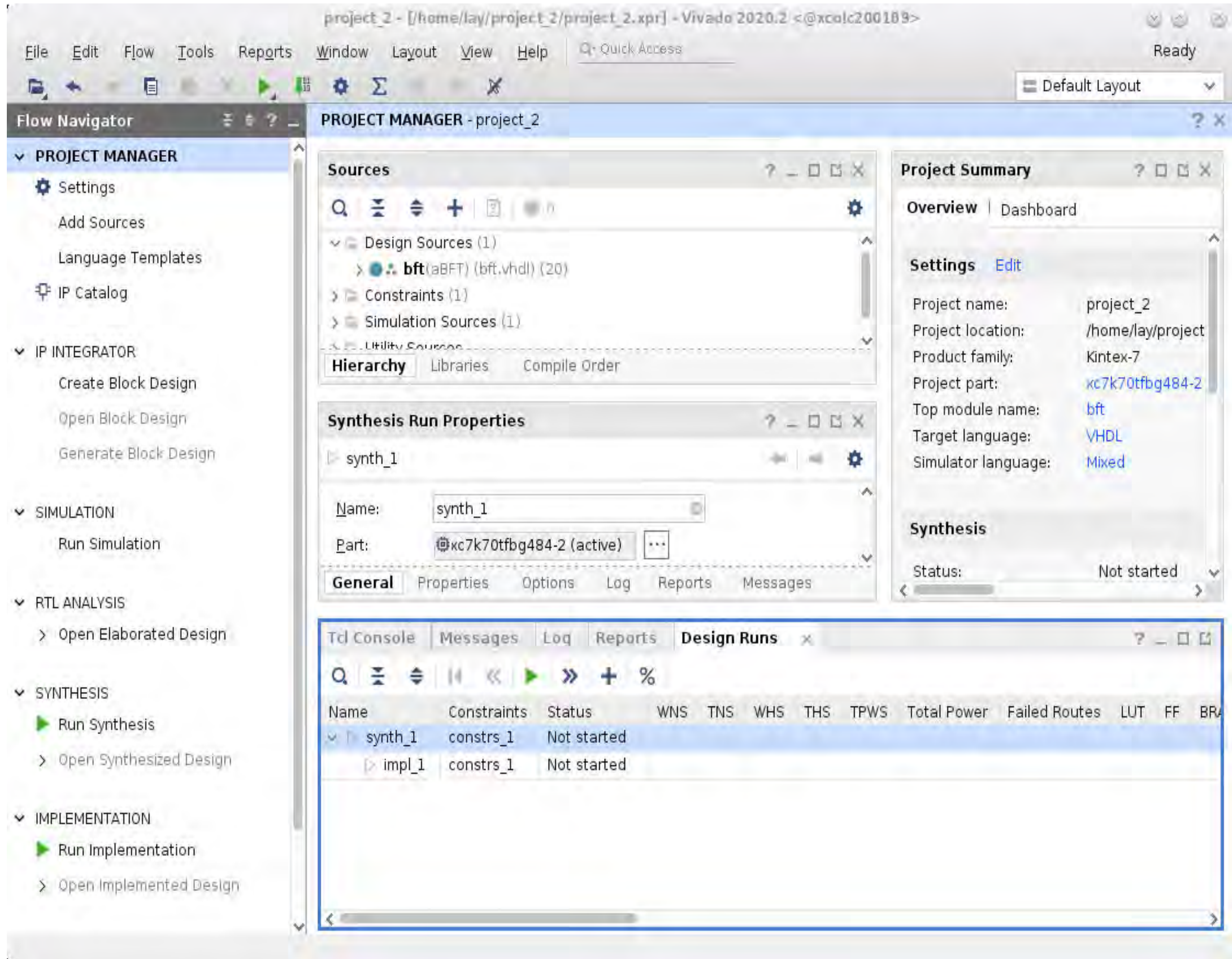


Figure A-5: Project Manager

f. In the launch runs window, choose "Lunch runs on cluster" and in the combo box, select the custom cluster name created above.

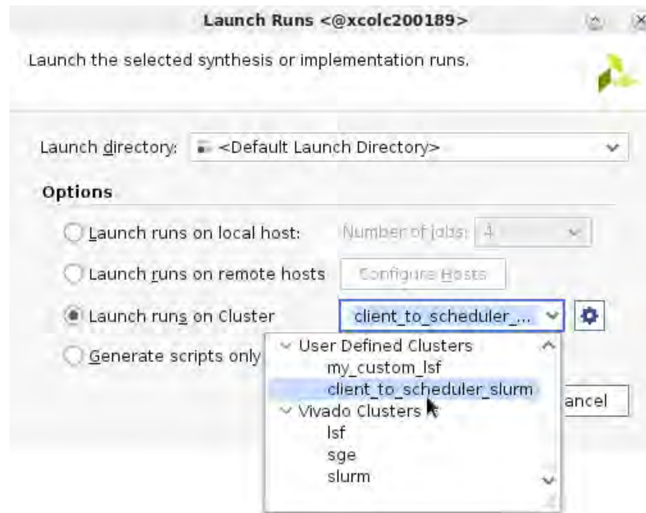


Figure A-6: Launch Runs

g. Click **OK** to launch the job.

h. In a terminal, `ssh` into the scheduler machine and check to see the job running using the `squeue` command on the scheduler machine.

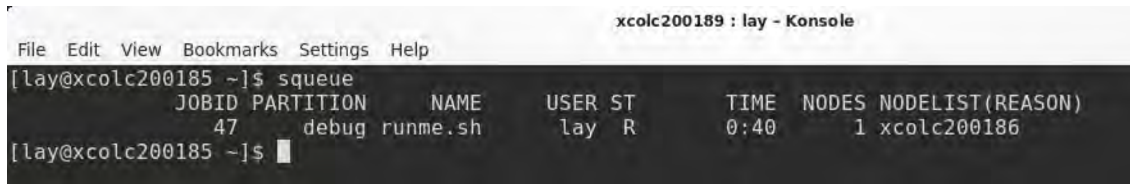


Figure A-7: Terminal Window

i. See the job complete successfully in the Vivado session running on the client.

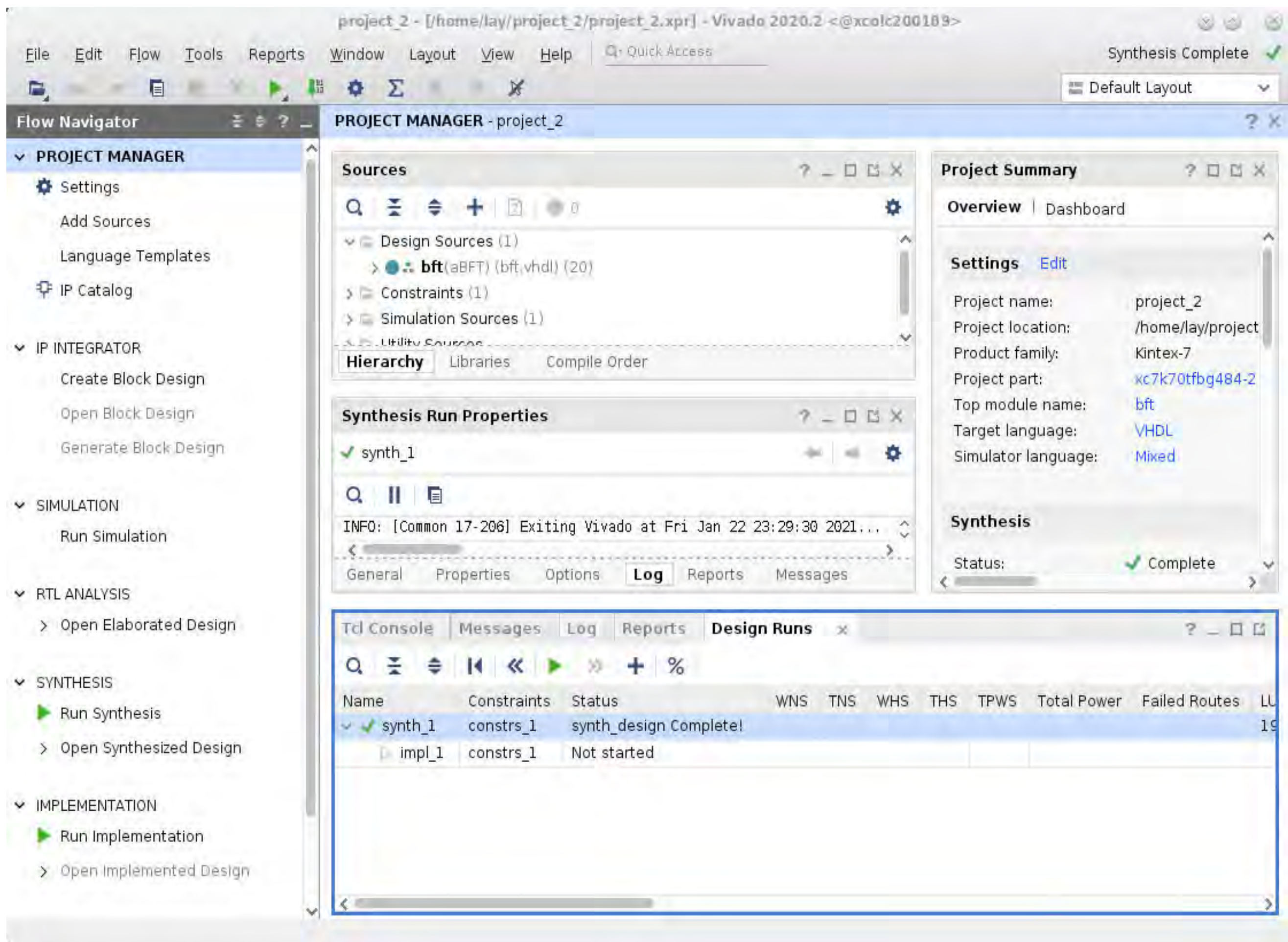


Figure A-8: Project Manager

Launching Jobs on Remote Hosts

Once remote hosts are configured, using them to launch Vivado jobs is easy. Figure A-9 shows the launch runs dialog box. When launching a run, choose either “Launch runs on remote hosts” or “Launch runs on cluster” and choose a specific cluster. The jobs will use your preconfigured settings to execute.

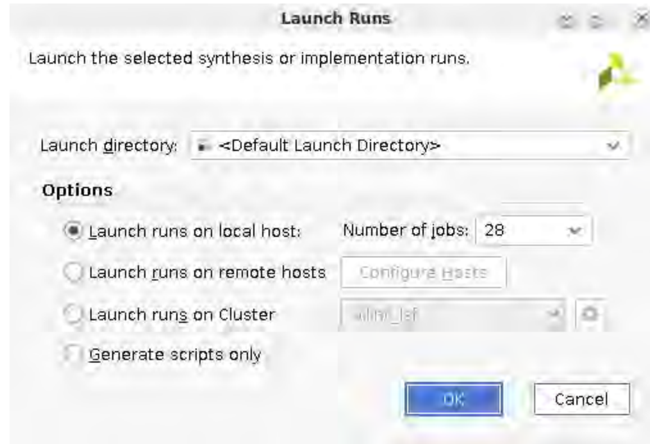


Figure A-9: Launch Runs Dialog Box.

Jobs can be executed on the user configured remote hosts or clusters.

ISE Command Map

Tcl Commands and Options

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

Table B-1 lists various ISE tool command line options, and their equivalent Vivado Design Suite Tcl command and Tcl command options. For more information about Tcl commands, see the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 19], 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 -mode out_of_context` (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. See Multithreading with the Vivado Tools, page 7 for details.
`map -ntd`	place_design `-non_timing_driven`
`map -power`	`power_opt_design`
`map -u`	`link_design -mode out_of_context`, `opt_design -retarget` (skip constant propagation and sweep)
`par -mt`	`route_design` automatically runs multi-threaded. See Multithreading with the Vivado Tools, page 7 for details.
`par -k`	The `route_design` command is always re-entrant.
`par -nopad`	The `-nopad` behavior is the Vivado tools default behavior. You must use `report_io` to obtain the `PAD` file report generated by PAR.
`par -ntd`	`route_design -no_timing_driven`

Appendix C

Implementation Categories, Strategy Descriptions, and Directive Mapping

Implementation Categories

Table C-1: **Implementation 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

Implementation Strategy Descriptions

Table C-2: **Implementation Strategy Descriptions**

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_ExplorePostRoutePhysOpt`	Similar to Peformance_Explore but adds `phys_opt_design` after routing for further improvements.
`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_EarlyBlockPlacement`	Finalize placement of Block RAM and DSPs in the early stages of global 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).

Table C-2: **Implementation Strategy Descriptions *(Cont'd)***

Implementation Strategy Name	Description
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_Retiming	Combines retiming in phys_opt_design with extra placement optimization and higher router delay cost.
Performance_ExtraTimingOpt	Runs additional timing-driven optimizations to potentially improve overall timing slack.
Performance_RefinePlacement	Increase placer effort in the post-placement optimization phase, and disable timing relaxation in the router.
Performance_SpreadSLL	A placement variation for SSI devices with tendency to spread SLR crossings horizontally.
Performance_BalanceSLL	A placement variation for SSI devices with more frequent crossings of SLR boundaries.
Congestion_SpreadLogic_high	Spread logic throughout the device to avoid creating congested regions (high setting is the highest degree of spreading).
Congestion_SpreadLogic_medium	Spread logic throughout the device to avoid creating congested regions (medium setting is the medium degree of spreading).
Congestion_SpreadLogic_low	Spread logic throughout the device to avoid creating congested regions (low setting is the lowest degree of spreading).
Congestion_SpreadLogic_Explore	Similar to Congestion_SpreadLogic_high, but uses the Explore directive for routing.
Congestion_SSI_SpreadLogic_high	Spread logic throughout the device to avoid creating congested regions, intended for SSI devices (high setting is the highest degree of spreading).
Congestion_SSI_SpreadLogic_low	Spread logic throughout the device to avoid creating congested regions, intended for SSI devices (low setting is the lowest degree of spreading).
Area_Explore	Uses multiple optimization algorithms to get potentially fewer LUTs.
Area_ExploreSequential	Similar to Area_Explore but adds optimization across sequential cells.
Area_ExploreWithRemap	Similar to Area_Explore but adds the remap optimization to compress logic levels.
Power_DefaultOpt	Adds power optimization (power_opt_design) to reduce power consumption.
Power_ExploreArea	Combines sequential area optimization with 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).
Flow_RunPostRoutePhysOpt	Similar to Flow_RunPhysOpt, but enables the Post-Route physical optimization step with the -directive Explore option.
Flow_RuntimeOptimized	Each implementation step trades design performance for better run time. Physical optimization (phys_opt_design) is disabled.
Flow_Quick	Fastest possible runtime, all timing-driven behavior disabled. Useful for utilization estimation.

Directives Used By opt_design and place_design in Implementation Strategies

Table C-3: **Directives Used by opt_design and place_design in Implementation Strategies**

Strategy	opt_design -directive	place_design -directive
Performance_Explore	Explore	Explore
Performance_ExplorePostRoutePhysOpt	Explore	Explore
Performance_ExploreWithRemap	ExploreWithRemap	Explore
Performance_WLBlockPlacement	Default	WLDrivenBlockPlacement
Performance_WLBlockPlacementFanoutOpt	Default	WLDrivenBlockPlacement
Performance_EarlyBlockPlacement	Explore	EarlyBlockPlacement
Performance_NetDelay_high	Default	ExtraNetDelay_high
Performance_NetDelay_low	Explore	ExtraNetDelay_low
Performance_Retiming	Default	ExtraPostPlacementOpt
Performance_ExtraTimingOpt	Default	ExtraTimingOpt
Performance_RefinePlacement	Default	ExtraPostPlacementOpt
Performance_SpreadSLLs	Default	SSI_SpreadSLLs
Performance_BalanceSLLs	Default	SSI_BalanceSLLs
Performance_BalanceSLRs	Default	SSI_BalanceSLRs
Performance_HighUtilSLRs	Default	SSI_HighUtilSLRs
Congestion_SpreadLogic_high	Default	AltSpreadLogic_high
Congestion_SpreadLogic_medium	Default	AltSpreadLogic_medium
Congestion_SpreadLogic_low	Default	AltSpreadLogic_low
Congestion_SSI_Spreadlogic_high	Default	SSI_SpreadLogic_high
Congestion_SSI_Spreadlogic_low	Default	SSI_SpreadLogic_low
Area_Explore	ExploreArea	Default
Area_ExploreSequential	ExploreSequentialArea	Default
Area_ExploreWithRemap	ExploreWithRemap	Default
Power_DefaultOpts	Default	Default
Power_ExploreArea	ExploreSequentialArea	Default
Flow_RunPhysOpt	Default	Default
Flow_RunPostRoutePhysOpt	Default	Default
Flow_RuntimeOptimized	RuntimeOptimized	RuntimeOptimized
Flow_Quick	RuntimeOptimized	Quick

Directives Used by phys_opt_design and route_design in Implementation Strategies

Table C-4: **Directives Used by phys_opt_design and route_design in Implementation Strategies**

Strategy	phys_opt_design -directive	route_design -directive
Performance_Explore	Explore	Explore
Performance_ExplorePostRoutePhysOpt	Explore[a]	Explore
Performance_ExploreWithRemap	Explore	NoTimingRelaxation
Performance_WLBlockPlacement	Explore	Explore
Performance_WLBlockPlacementFanoutOpt	AggressiveFanoutOpt	Explore
Performance_EarlyBlockPlacement	Explore	Explore
Performance_NetDelay_high	AggressiveExplore	NoTimingRelaxation
Performance_NetDelay_low	AggressiveExplore	NoTimingRelaxation
Performance_Retiming	AlternateFlowWithRetiming	Explore
Performance_ExtraTimingOpt	Explore	NoTimingRelaxation
Performance_RefinePlacement	Default	NoTimingRelaxation
Performance_SpreadSLLs	Explore	Explore
Performance_BalanceSLLs	Explore	Explore
Performance_BalanceSLRs	Explore	Explore
Performance_HighUtilSLRs	Explore	Explore
Congestion_SpreadLogic_high	AggressiveExplore	AlternateCLBRouting
Congestion_SpreadLogic_medium	Explore	AlternateCLBRouting
Congestion_SpreadLogic_low	Explore	AlternateCLBRouting
Congestion_SSI_SpreadLogic_high	AggressiveExplore	AlternateCLBRouting
Congestion_SSI_SpreadLogic_low	Explore	AlternateCLBRouting
Area_Explore	Not enabled	Default
Area_ExploreSequential	Not enabled	Default
Area_ExploreWithRemap	Not enabled	Default
Power_DefaultOpts	Not enabled	Default
Power_ExploreArea	Not enabled	Default
Flow_RunPhysOpt	Explore	Default
Flow_RunPostRoutePhysOpt	Explore[a]	Default
Flow_RuntimeOptimized	Not enabled	RuntimeOptimized
Flow_Quick	Not enabled	Quick

a. Explore applies to both post-place and post-route `phys_opt_design`

Listing the Strategies for a Release

You can list the Synthesis and Implementation Strategies for a particular release using the `list_property_value` command in an open Vivado project. The following are examples using a Vivado version 2017.3 project containing synthesis run `synth_1` and implementation run `impl_1`.

```
Vivado% join [list_property_value strategy [get_runs synth_1] ] \n
Vivado Synthesis Defaults
Flow_AreaOptimized_high
Flow_AreaOptimized_medium
Flow_AreaMultThresholdDSP
Flow_AlternateRoutability
Flow_PerfOptimized_high
Flow_PerfThresholdCarry
Flow_RuntimeOptimized

Vivado% join [list_property_value strategy [get_runs impl_1] ] \n
Vivado Implementation Defaults
Performance_Explore
Performance_ExplorePostRoutePhysOpt
Performance_WLBlockPlacement
Performance_WLBlockPlacementFanoutOpt
Performance_EarlyBlockPlacement
Performance_NetDelay_high
Performance_NetDelay_low
Performance_Retiming
Performance_ExtraTimingOpt
Performance_RefinePlacement
Performance_SpreadSLLs
Performance_BalanceSLLs
Congestion_SpreadLogic_high
Congestion_SpreadLogic_medium
Congestion_SpreadLogic_low
Congestion_SpreadLogic_Explore
Congestion_SSI_SpreadLogic_high
Congestion_SSI_SpreadLogic_low
Area_Explore
Area_ExploreSequential
Area_ExploreWithRemap
Power_DefaultOpt
Power_ExploreArea
Flow_RunPhysOpt
Flow_RunPostRoutePhysOpt
Flow_RuntimeOptimized
Flow_Quick
```

The list of strategies also includes user-defined strategies.

Listing the Directives for a Release

You can display the list of directives for a command for a particular release. This is done programmatically using Tcl to list the properties of the runs. Each design run has a property corresponding to a Design Runs step command:

```
STEPS.<STEP>_DESIGN.ARGS.DIRECTIVE
```

Where `<STEP>` is one of `SYNTH`, `OPT`, `PLACE`, `PHYS_OPT`, or `ROUTE`. This property is an enum type, so all supported values can be returned using `list_property_value`. Following is an example:

```
Vivado% list_property_value STEPS.SYNTH_DESIGN.ARGS.DIRECTIVE [get_runs synth_1]
RuntimeOptimized
AreaOptimized_high
AreaOptimized_medium
AlternateRoutability
AreaMapLargeShiftRegToBRAM
AreaMultThresholdDSP
FewerCarryChains
Default
```

The following Tcl example shows how to list the directives for each synthesis and implementation command using a temporary, empty project:

```
create_project p1 -force -part xcku035-fbva900-2-e

#get synth_design directives
set steps [list synth]
set run [get_runs synth_1]
foreach s $steps {
  puts "${s}_design Directives:"
  set dirs [list_property_value STEPS.${s}_DESIGN.ARGS.DIRECTIVE $run]
  set dirs [regsub -all {\s} $dirs \n]
  puts "$dirs\n"
}

#get impl directives
set steps [list opt place phys_opt route]
set run [get_runs impl_1]
foreach s $steps {
  puts "${s}_design Directives:"
  set dirs [list_property_value STEPS.${s}_DESIGN.ARGS.DIRECTIVE $run]
  set dirs [regsub -all {\s} $dirs \n]
  puts "$dirs\n"
}
close_project -delete
```

XILINX

Appendix D

Additional Resources and Legal Notices

Xilinx Resources

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

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.

Documentation Navigator and Design Hubs

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

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

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

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

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

References

Vivado Design Suite User Guides

1. *Vivado Design Suite User Guide: Design Flows Overview* (UG892)
2. *Vivado Design Suite User Guide: Hierarchical Design* (UG905)
3. *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893)
4. *Vivado Design Suite User Guide: Designing with IP* (UG896)
5. *Vivado Design Suite User Guide: Using Tcl Scripting* (UG894)
6. *Vivado Design Suite User Guide: System-Level Design Entry* (UG895)
7. *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994)
8. *Vivado Design Suite User Guide: Synthesis* (UG901)
9. *Vivado Design Suite User Guide: Using Constraints* (UG903)
10. *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906)
11. *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907)
12. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
13. *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949)
14. *Vivado Design Suite Properties Reference Guide* (UG912)
15. *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909)
16. *Versal ACAP Clocking Resources Architecture Manual* (AM003)

Other Vivado Design Suite Documents

17. *7 Series FPGAs Clocking Resources User Guide* (UG472)
18. *UltraScale™ Architecture Clocking Resources Advanced Specification User Guide* (UG572)
19. *Vivado Design Suite Tcl Command Reference Guide* (UG835)
20. *Vivado Design Suite Migration Guide* (UG911)
21. *Vivado Design Suite Tutorial: Design Flows Overview* (UG888)

Vivado Design Suite Documentation Site

22. Vivado Design Suite Documentation

Training Resources

Xilinx provides a variety of training courses and QuickTake videos to help you learn more about the concepts presented in this document. Use these links to explore related training resources:

1. Designing FPGAs Using the Vivado Design Suite 1
2. Designing FPGAs Using the Vivado Design Suite 2
3. Designing FPGAs Using the Vivado Design Suite 3
4. Designing FPGAs Using the Vivado Design Suite 4

Please Read: Important Legal Notices

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

AUTOMOTIVE APPLICATIONS DISCLAIMER

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

© Copyright 2012-2021 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.