Vivado Design Suite User Guide

Implementation

UG904 (v2025.2) November 20, 2025

AMD

Table of Contents

Chapter 1

Preparing for Implementation

About the Vivado Implementation Process

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

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

The Figure 1: Vivado Design Suite High-Level Design Flow shows the Vivado tools flow.

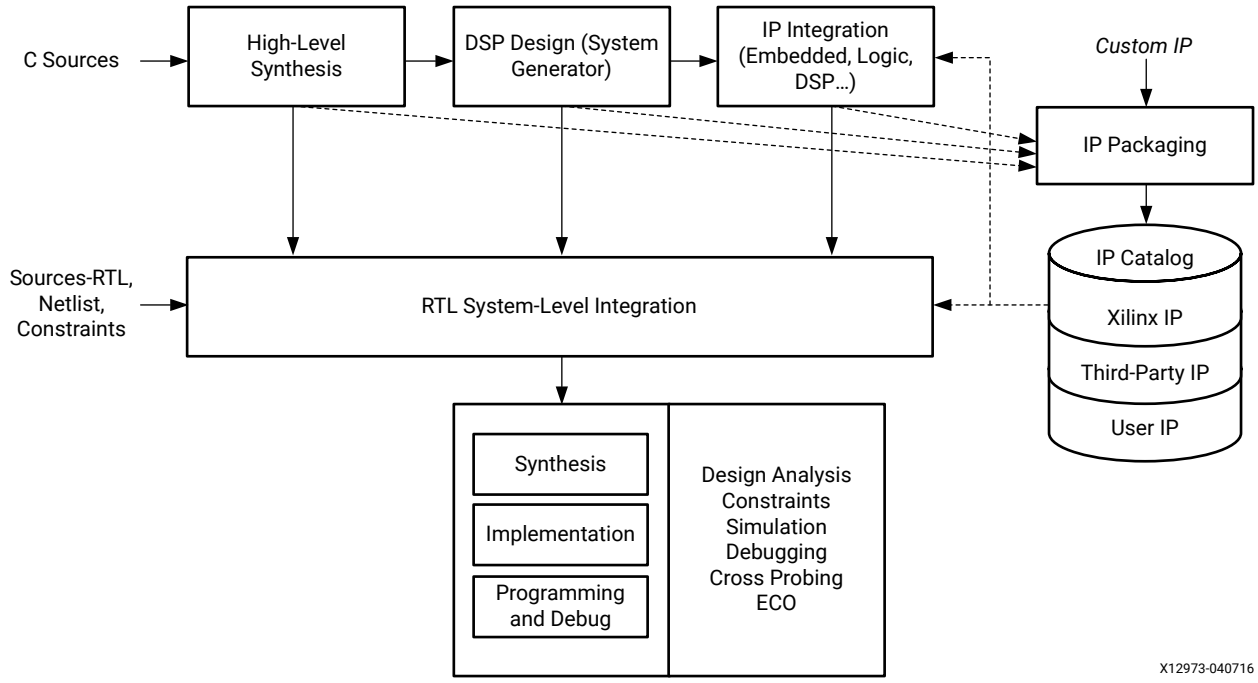
Vivado implementation includes all steps necessary to place and route the netlist onto device resources, within the logical, physical, and timing constraints of the design.

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

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: **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 AMD device.
2. Power Opt Design (optional): Optimizes design elements to reduce the power demands of the target AMD device.
3. Place Design: Places the design onto the target AMD 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 AMD device.
7. Post-Route Phys Opt Design (optional): Optimizes logic, placement, and routing using actual routed delays.
8. Write Bitstream: Generates a bitstream for AMD device configuration. Typically, bitstream generation follows implementation.

For more information about writing the bitstream, see Generating the Bitstream or Device Image in the *Vivado Design Suite User Guide: Programming and Debugging* (UG908).

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 maximum number of simultaneous threads depends on the OS. For Windows systems, the limit is 2; for Linux systems the default is 8. You can change the limit by 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
```

All tasks are limited to two threads, regardless of processor count or task you execute. If the system has at least eight processors, you can set the limit to 8. You can also 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

Parallel Runs

Vivado supports launching design runs in parallel by providing the launch_runs -jobs option to specify the number of simultaneous runs. Each simultaneous run is an independent process, requiring its own CPU and memory resources.

It is important to allocate sufficient resources to handle the total peak computing requirements. For example, consider a design run that typically reports a peak usage of 20 GB RAM with general.maxThreads set to 8. Launching four similar runs in parallel requires 32 processor cores. Additionally, it requires roughly 80 GB RAM to avoid performance degradation due to competition for computing resources by the four processes.

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) or type `<command> -help`.

Navigating Content by Design Process

AMD Adaptive Computing documentation is organized around a set of standard design processes to help you find relevant content for your current development task. You can access the AMD Versal™ adaptive SoC design processes on the Design Hubs page. You can also use the Design Flow Assistant to better understand the design flows and find content that is specific to your intended design needs. This document covers the following design processes:

- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, functional simulation, and evaluating the AMD 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. The design must pass through implementation to generate a bitstream that you can download onto an AMD device.

Implementation is a series of steps that takes the logical netlist and maps it into the physical array of the target AMD 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, Vivado creates an on-disk directory structure to organize design sources, run results, reports, and project status.

Vivado stores the project infrastructure that enables automated management of design data, processes, and status in the 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.

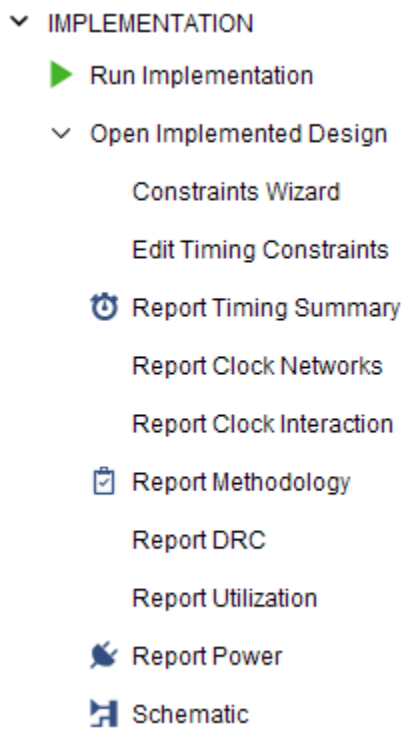
To run implementation in Project Mode, you click the Run Implementation button in the IDE or use the `launch_runs` Tcl command. See Using Project Mode in the *Vivado Design Suite User Guide: Design Flows Overview* (UG892) for information about using projects in the Vivado Design Suite.

Flow Navigator

The Vivado Integrated Design Environment (IDE) integrates the complete design flow. 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. The following figure shows the Implementation section of the Flow Navigator.

Figure 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 design flow, see the Vivado Design Suite User Guide: Using the Vivado IDE (UG893).*

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 steps 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, if you fail to write a checkpoint before exiting Vivado, the implementation run needs to be rerun to get back to the same design state before exiting. For this reason, AMD 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

You can run Vivado implementation in either Project Mode or Non-Project Mode. Also, you can use the Vivado IDE and Tcl API 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 the Non-Project Mode does not support. 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.

This guide describes how to run implementation in Project Mode and Non-Project Mode.

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)
- *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893)

Beginning the Implementation Flow

The implementation flow typically begins by loading a synthesized design into memory. You can run the implementation flow, analyze and refine the design and constraints, and reload the design after updates.

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

- Run Vivado synthesis. In Project Mode, the synthesis run produces results that Vivado automatically uses as input to the 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. you can use synthesized netlists 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.

Load the synthesized design without running implementation to analyze and refine constraints.

- 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 can include placement and routing data. This enables reentrant implementation flows: load a routed design and edit routing, or load a placed design and run multiple routes with different options.

Importing Previously Synthesized Netlists

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

- Structural Verilog
- Structural SystemVerilog
- EDIF
- AMD NGC
- Synthesized Design Checkpoint (DCP)

> **IMPORTANT!** *Vivado Design Suite does not support NGC format files for UltraScale and later devices. 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, AMD 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 consistent use of IP output products throughout the design flow. If the IP was synthesized out of context and has an associated DCP, Vivado automatically uses it and does not re-synthesize the IP.*
>
> For more information, see section Adding Existing IP to a Project in the *Vivado Design Suite User Guide: Designing with IP* (UG896).

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).

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).

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.

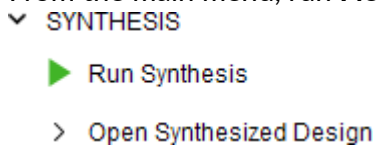
For more information, see Opening the Synthesized Design.

Loading the Design Netlist in Project Mode Before Implementation

In Project Mode, after an RTL synthesis, 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.

Advanced Flow

The Advanced Flow is applicable for all Versal families. It does not affect 7 series or UltraScale architectures. The Advanced Flow is a new place and route feature set. It enables both faster compile times and improved performance by leveraging more advanced technologies and parallelization techniques.

The following captures some key changes in the implementation flow due to the introduction of the Advanced Flow set of features.

Table 1: **Advanced Flow**

Feature	Impact
Implementation Runs	Faster compile times. Improved design performance. Log file differences.
Place Design directives, subdirectives and tool options	Place Design allows both directive level selection and sub directive selection enabling fine grain selection of one or more components that make up a directive.
Place Design Default VTree	Changed from InterSLR to Balanced.
ECO	Standard placer is used instead of incremental placer.
Intelligent Design Runs	Not supported. Use Automatic QoR Suggestion flow instead.
ML Strategy	Not supported. Run multiple strategies or place_design directives instead.
Auto directives	Not supported. Run multiple strategies or place_design directives instead.
Incremental Implementation	Not supported
Power Opt Design	Not supported

For more information, refer to the following pages:

- place_design for Versal
- Appendix B: Implementation Categories, Strategy Descriptions, and Directive Mapping
- Automatic QoR Suggestions

Migrating to the Advanced Flow

You can migrate Versal designs created before 2024.2 to the Advanced Flow. Upgrading to the Advanced Flow requires some extra considerations than when compared to a typical project upgrade due to a Vivado version change. You might *not* want to migrate designs in production or designs nearing completion, especially if you have invested many iterations to improve timing. Or if you depend on features such as Intelligent Design Runs that the Advanced Flow does not support.

You can not reuse place and route data from the 2024.1 and earlier Vivado versions in the Advanced Flow. This impacts DFX platform designs where the static logic is created using a pre-2024.2 version of Vivado. Refer to *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909) for more details about DFX designs with Advanced Flow.

For the latest information about Advanced Flow migration, check Answer Record 000036830.

For Vivado projects, automatic migration takes place when you open a Vivado project in a later Vivado version. The migration process modifies the project's implementation runs to ensure to use only Advanced Flow–supported commands. After migrating a Versal project to Advanced Flow, you cannot open it in earlier Vivado versions. Therefore, back up the project under revision control. For non-project users, modify minor scripts. The areas modified are captured below:

Table 2: **Migration to the Advanced Flow**

Feature	Project Flow Impact	Non Project Impact Commands
Implementation Runs	Implementation runs are reset Strategy is mapped to equivalent strategy if it exists, else default	Modify place_design command line: place_design -directive <..> -net_delay_weight <..> -subdirective <..>
Custom run strategy	Mapped to default strategy	
Place Design Default VTree	Changed from InterSLR to Balanced	place_design -clock_vtree_type <..>
Intelligent Design Runs	Run is deleted	n/a
Incremental Implementation	Mapped to default strategy	Remove the following commands: read_checkpoint -incremental report_incremental_reuse

Table 2: **Migration to the Advanced Flow** *(cont'd)*

Feature	Project Flow Impact	Non Project Impact Commands
ML Strategy	Mapped to default strategy	Remove commands with the following directives: opt_design -directive RQS place_design -directive RQS phys_opt_design -directive RQS route_design -directive RQS
Auto directives	Mapped to default strategy	Remove commands with the following directives: place_design -directive Auto_1, Auto_2, and Auto_3
Power Opt Design	Power opt design is disabled as part of strategy migration	Remove the following commands: power_opt_design report_power_opt set_power_opt
ECO	Incremental placer is no longer used. Standard place is used instead with eco switch. No change to route_design for ECO	New command to use: place_design -eco -no_timing_driven

For more information, refer to the following documents:

- Appendix B: Implementation Categories, Strategy Descriptions, and Directive Mapping
- place_design for Versal
- Vivado ECO Flow

Configuring, Implementing, and Verifying IP

For information on importing IP into your design prior to synthesis, see section Creating an IP Customization in the *Vivado Design Suite User Guide: Designing with IP* (UG896).

Guiding Implementation with Design Constraints

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

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 design frequency requirements and are written in Xilinx Design Constraints (XDC), which is based on the industry-standard SDC.

Without timing constraints, the Vivado Design Suite optimizes the design solely for wire length and routing congestion, and does not 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 the *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907).

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 Migrating UCF Constraints to XDC in the *ISE to Vivado Design Suite Migration Guide* (UG911).

Constraint Sets Apply Lists of Constraint Files to Your Design

A constraint set is a list of constraint files that you can apply 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 make your constraint strategy clearer, and to support 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 AMD devices for the same project. You may need different physical and timing constraints 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 affecting placement and routing, see Physical Constraint in the *Vivado Design Suite User Guide: Using Constraints* (UG903).

Adding Constraints as Attribute Statements

You can add constraints to HDL sources as attribute statements. Add attributes 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, define Relatively Placed Macros (RPMs) 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 section Migrating UCF Constraints to XDC in the *Vivado Design Suite User Guide: Using Constraints* (UG903).

For more information on constraints that XDC does not support, see the *ISE to Vivado Design Suite Migration Guide* (UG911).

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

You can run through the checkpoint designs through the remainder of the design flow using Tcl commands. You cannot modify them 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, you must rerun the flow to get back to the same stage.*

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. Open the design checkpoint as a separate in-memory design.

The related Tcl command is `open_checkpoint`.

Optimizing Compile Time

AMD Vivado™ Design Suite compile time is influenced by multiple factors, including the complexity of the design netlist, device utilization, physical and timing constraints, and the strategies used for implementation commands. Additionally, the number of intermediate reports, the number of threads used by the Vivado tools, the configuration of the machine running the Vivado tools, and the load sharing facility (LSF) job request settings all play significant roles. The following sections examine these factors and provide recommendations to optimize and reduce compile time.

Profiling Compile Time

To compare the compile time of Vivado Design Suite commands across multiple phases on multiple designs, use the VivadoRuntime script to extract debug information from the compile time, memory, checksum, timing, congestion, and command lines inside a `vivado.log` file. To debug the compile time issue of a specific design in the Vivado Design Suite Tcl shell based on executing the targeted operations, use the `profiler.tcl` profiler file. For more information, see the Saving Compile Time Series 4: Profiling Compile Time with Tcl scripts blog.

Design and Constraints Impact on Compile Time

The following factors impact the compile time:

- Netlist complexity and utilization
- Timing constraints and optimization

You can use the following Vivado tools features to reduce compile time:

- **Multi-threading:** On multiprocessor systems, the Vivado tools use multi-threading to speed up certain processes, including DRC reporting, static timing analysis, placement, and routing..
- **Generate Parallel Reports:** Allows you to generate certain reports in parallel to reduce compile time. For more information, see the generate_parallel_reports command in the *Vivado Design Suite Tcl Command Reference Guide* (UG835).
- **Quick Directive:** Sets the absolute, fastest compile time for place and route. This directive applies to both `place_design` and `route_design`, is non-timing-driven, and performs the minimum compilation needed to place and route a design.

In addition, the performance of the compute host server is important in achieving efficient compile times. Optimizing the BIOS and CPU settings is essential for maximizing this performance. The following table outlines recommended settings for CPUs and servers, providing guidance on how to configure your system for optimal operation.

Table 3: **Recommended Settings for CPUs and Servers**

Setting	Recommendation
Simultaneous Multithreading (SMT) or Hyper-Threading	Enable to allow each core to handle two threads.
Turbo Boost/Precision Boost	Enable to allow dynamic clock speed increases.
Power Management	Set to prioritize performance over energy savings.
Memory Configuration	Enable to run RAM at its rated speed for maximum performance.
NUMA	Enable for optimized memory access in multi-socket systems.
C-States	Consider disabling deeper sleep C-states to reduce latency.
Virtualization	Disable if not using virtualization to reduce overhead.

Related Information

Multithreading with the Vivado Tools

Implementing the Design

Running Implementation in Non-Project Mode

To implement the synthesized design or netlist onto the targeted AMD devices in Non-Project Mode, 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 AMD device.
- **Power Opt Design (`power_opt_design`) (optional):** Optimizes design elements to reduce the power demands of the target AMD device.
- **Place Design (`place_design`):** Places the design onto the target AMD 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 AMD 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 AMD device configuration except for AMD Versal™ adaptive SoCs. Typically, bitstream generation follows implementation.
- **Write Device Image (`write_device_image`):** Generates a programmable device image for programming a Versal device.

For more information about writing the bitstream or creating a device image, see section Generating Bitstream in the *Vivado Design Suite User Guide: Programming and Debugging* (UG908).

These steps are collectively known as implementation. Enter the commands in any of the following ways:

- In the Tcl Console from the AMD 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 following script is an example of running implementation in Non-Project Mode. Assuming the script is named `run.tcl`, 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` must 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 are as follows:

- 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: Perform Physical Optimization
- Step 7: Route the Design
- Step 8: Run Required Reports
- Step 9: 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 let Vivado read a file on the disk and build the in-memory design without copying the file or creating a dependency on it.

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 AMD 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 IDE for interaction with the design data in a graphical form. tools, whether running in batch mode, Tcl shell mode for interactive Tcl commands, or in the

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.

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.

Step 5: Place the Design

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

Step 6: Perform Physical Optimization

Physical optimization performs timing-driven optimization on the negative-slack paths of a design. Many different types of optimizations are available. For more information see Physical Optimization.

Step 7: Route the Design

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

Step 8: 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 *Vivado Design Suite Tcl Command Reference Guide* (UG835) for further information on the `report_timing_summary` command and on the `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.

Step 9: 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
- AMD part data
- Placement and routing information

In Non-Project Mode, the design checkpoint file saves the design and allows it to reload 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

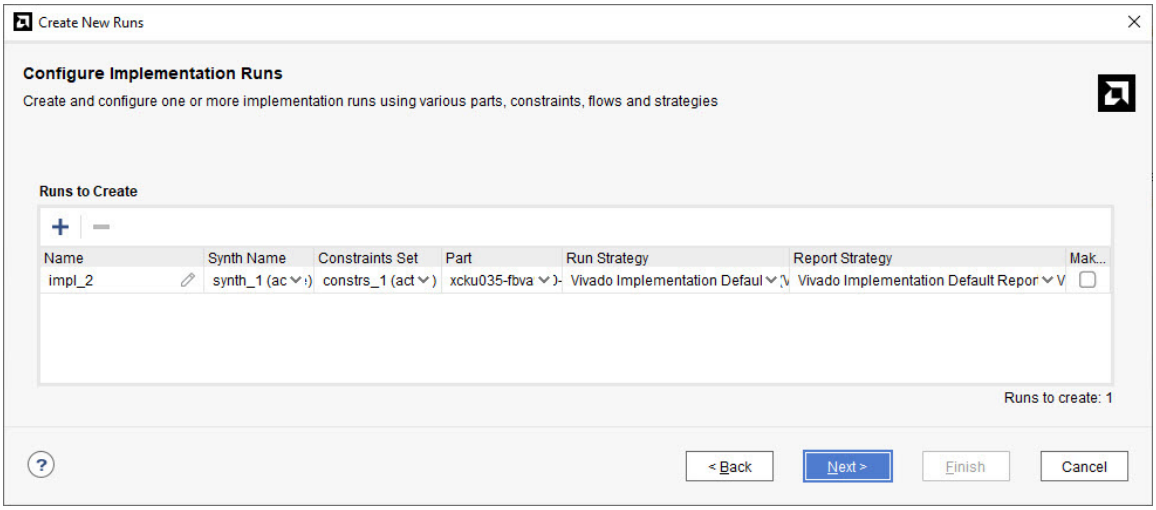
To define an implementation run:

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 page appears, as shown in the following figure. Specify settings as described in the steps below the figure.



   a. In the Name column, enter a name for the run 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 B: 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).

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).

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 section Configuring Project Settings in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895).

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

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 B: 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 B: Implementation Categories, Strategy Descriptions, and Directive Mapping.

   For more information see Defining Implementation Strategies.

   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 compile time. You must 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. The Performance_Explore strategy is a good first choice if your design goals cannot be met, and the increased runtime is acceptable. It covers all design types.

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

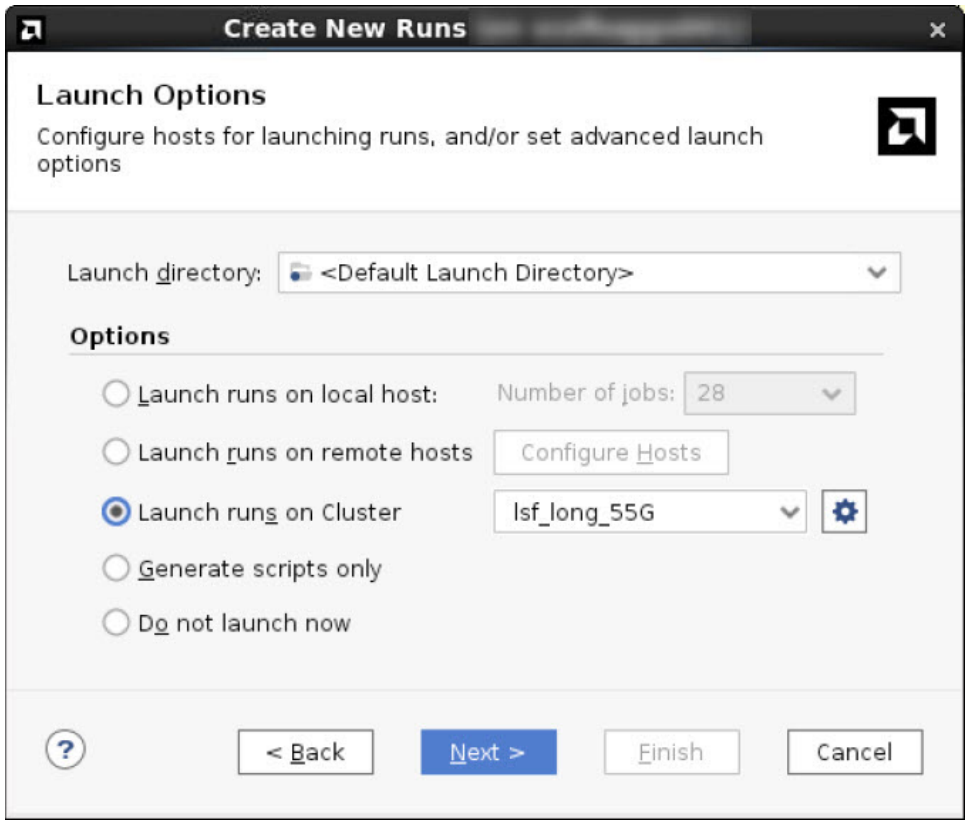
TIP: *Before launching a run, you can change the settings for each step in the implementation process, overriding the default settings for the selected strategy. You can also save those new settings as a new strategy. For more information, see Changing Implementation Run Settings.*

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.

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

h. Click **Next**.

4. The Launch Options page appears, as shown in the following figure. Specify options as described in the steps below the figure.



Note: The Launch runs on remote hosts and Launch runs on Cluster options shown in the previous figure 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 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) 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) 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 Define Cluster that has been added previously.
   - Select Generate scripts only to export and create the run directory and run script without launching the run script at this time. The script can be run later outside the Vivado IDE tools.
   - Select Do not launch now 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

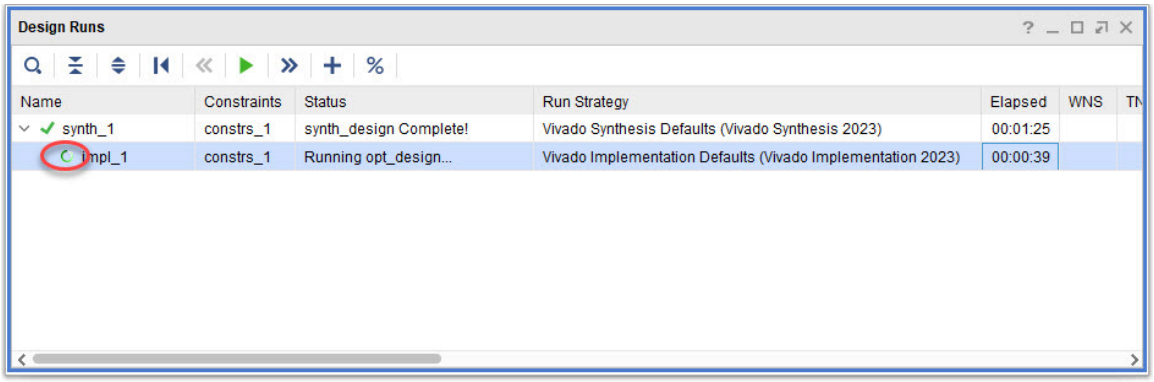
Select **Window** → **Design Runs** to open the Design Runs window.

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 section Using Data Table Windows in the *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893).

Figure 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.

Run Times

The Design Runs window reports start and elapsed run times.

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 window, Report window, 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

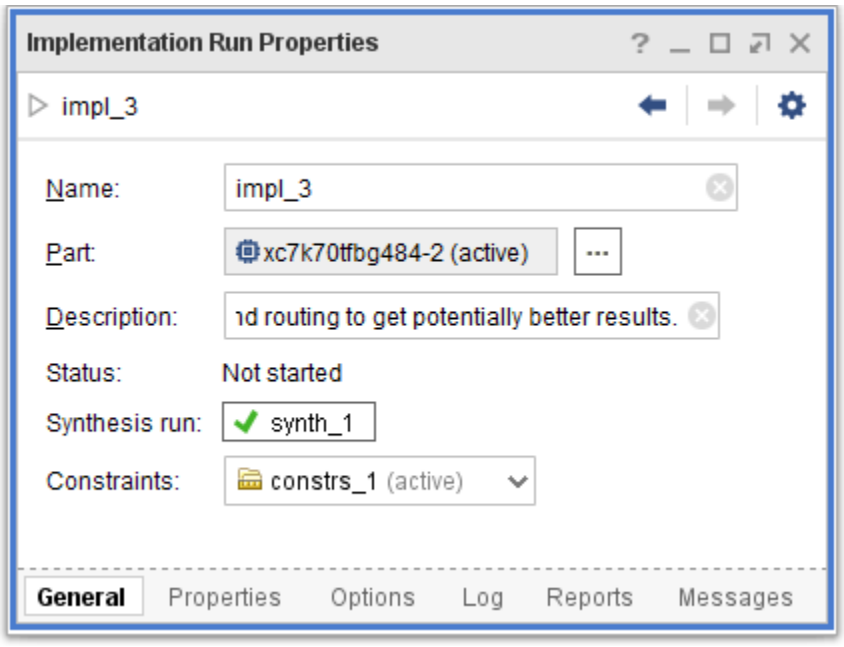
Select a run in the Design Runs window to display the current configuration of the run in the Run Properties window.

In the Run Properties window, you can change:

- The name of the run
- The AMD 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 section Using the Run Properties Window in the *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893).

Figure 4: **Implementation Run Properties Window**



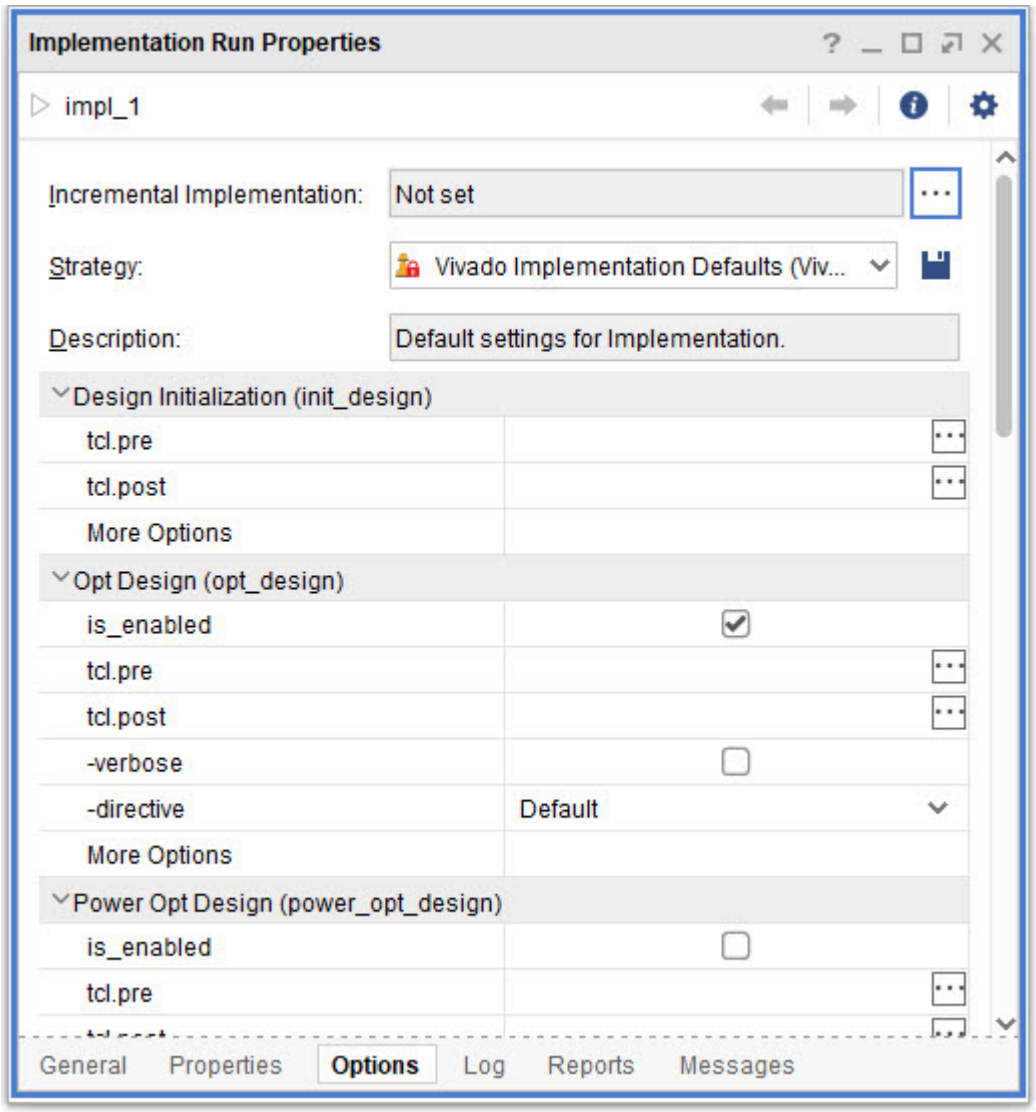
Specifying Design Run Settings

Specify design run settings in the Design Run Settings dialog box, shown in the following figure. 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 the following figure.

> **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.*

Figure 5: **Design Run Settings Dialog Box**



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.

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 *Vivado Design Suite Tcl Command Reference Guide* (UG835).

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 applied 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 the following figure).

- 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.

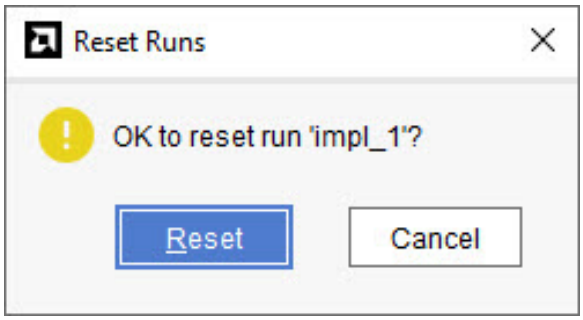
Resetting Runs

To reset a run:

1. Select a run in the Design Runs window.
2. Right-click and 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.

The Vivado tools prompt you to confirm the Reset Runs command, and optionally delete the generated files from the run directory. See the following figure.



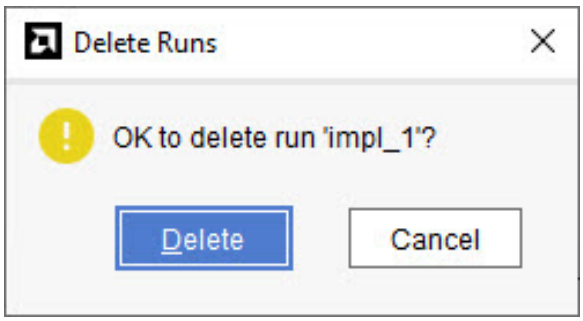
TIP: *The default setting is to delete the generated files. Disable this check box to preserve the generated run files.*

Deleting Runs

To delete runs from the Design Runs window:

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

As shown in the following figure, the Vivado tools prompt you to confirm the Delete Runs command.



Customizing Implementation Strategies

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

Figure 6: Implementation Settings shows the Implementation page in the Settings dialog box. To open this dialog box from the Vivado IDE, select **Tools → Settings** from the main menu.

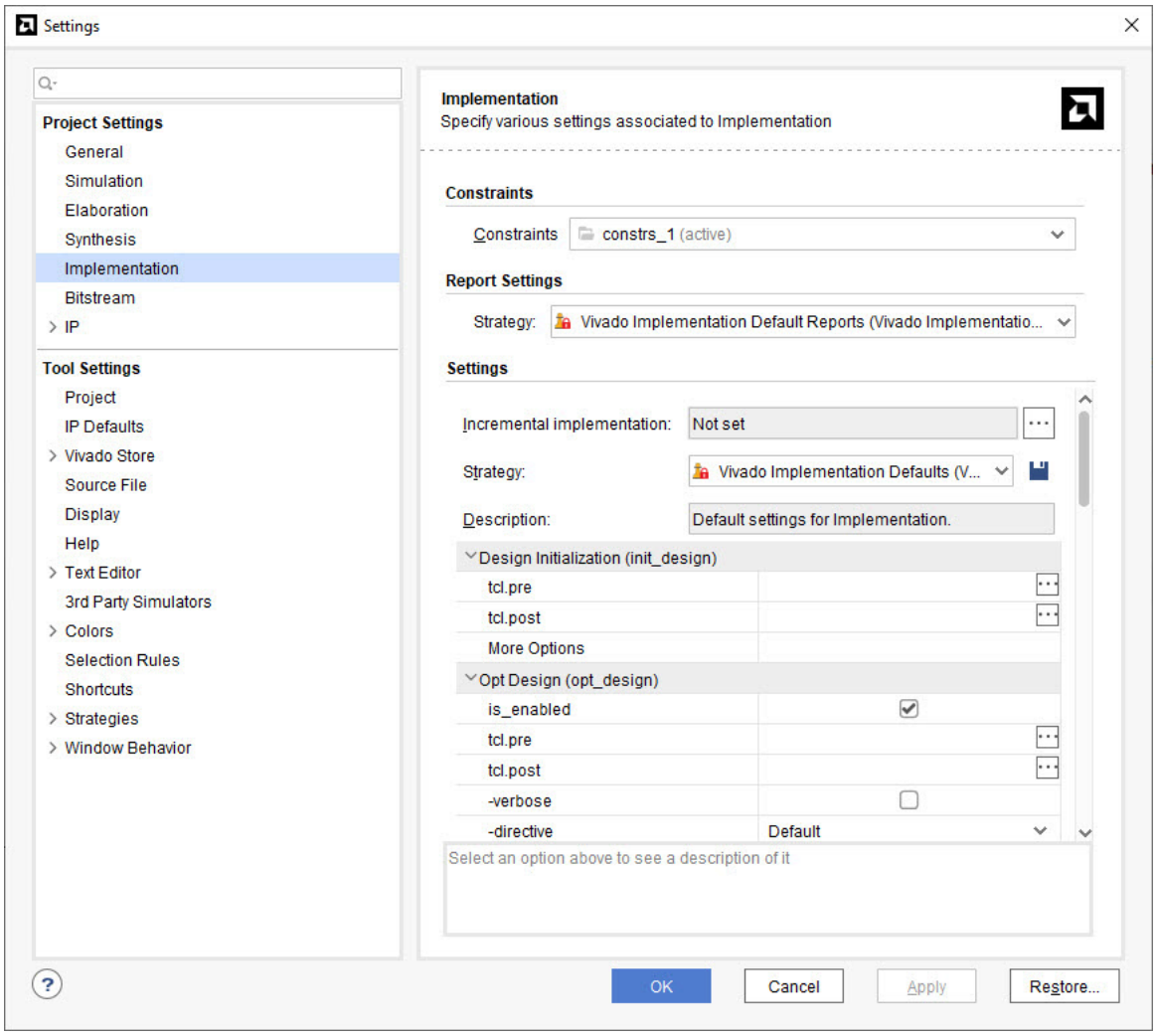
TIP: *The Settings command is not available in the Vivado IDE when running in non-project mode. In this case, you can define and preserve implementation strategies as Tcl scripts that can be used in batch mode, or interactively in the Vivado IDE.*

Accessing Implementation Settings for the Active Run from Flow Navigator

You can access Implementation Settings for the active implementation run by selecting **Settings** at the top of the Flow Navigator, then clicking the **Implementation** category. The Settings dialog box, shown in the following figure, 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 Implementation:** 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. You can not change the description of Vivado tools standard implementation strategies.

Figure 6: **Implementation Settings**

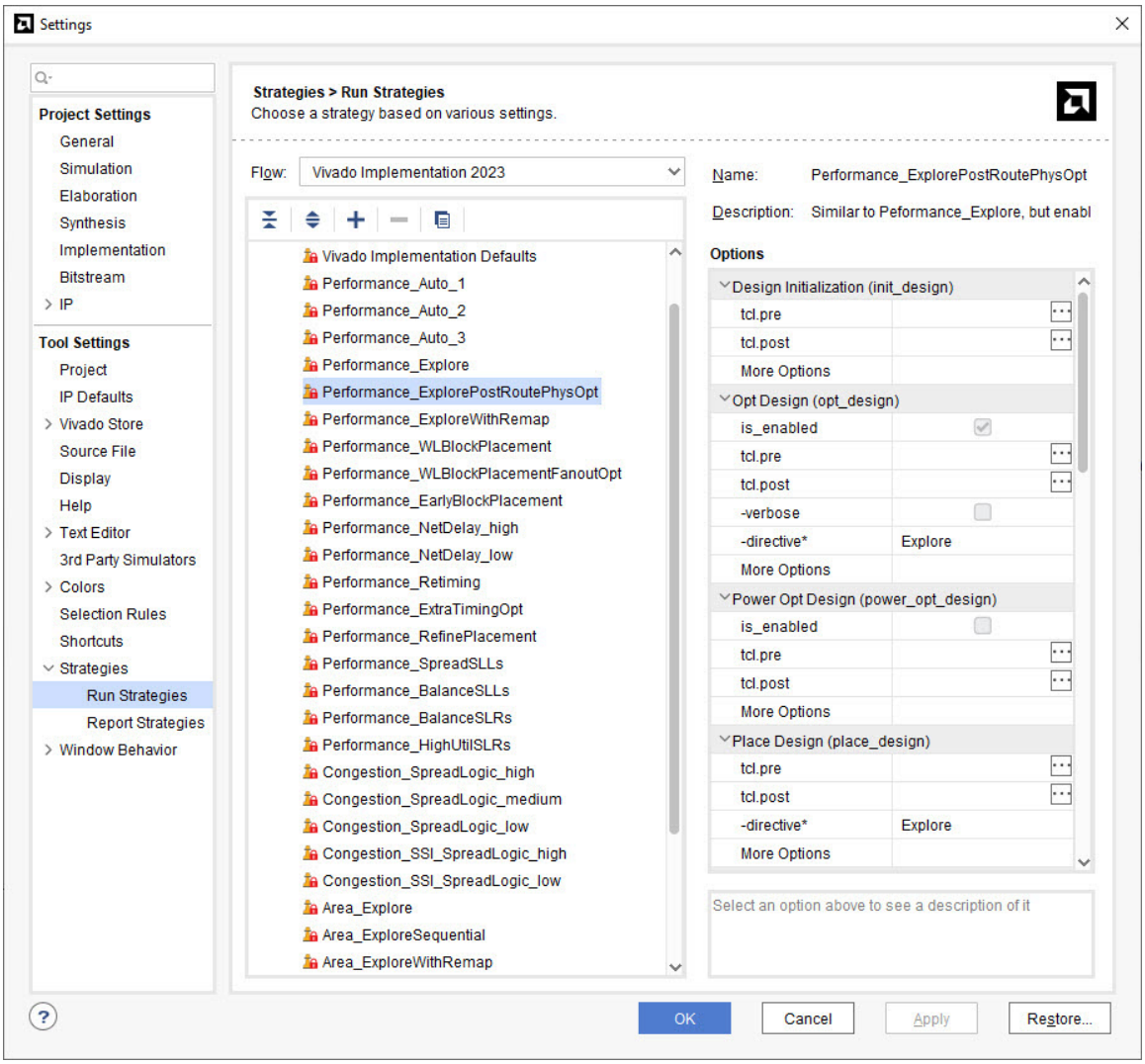


Defining Implementation Strategies

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 7: **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 a predefined strategy, then modify and save 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

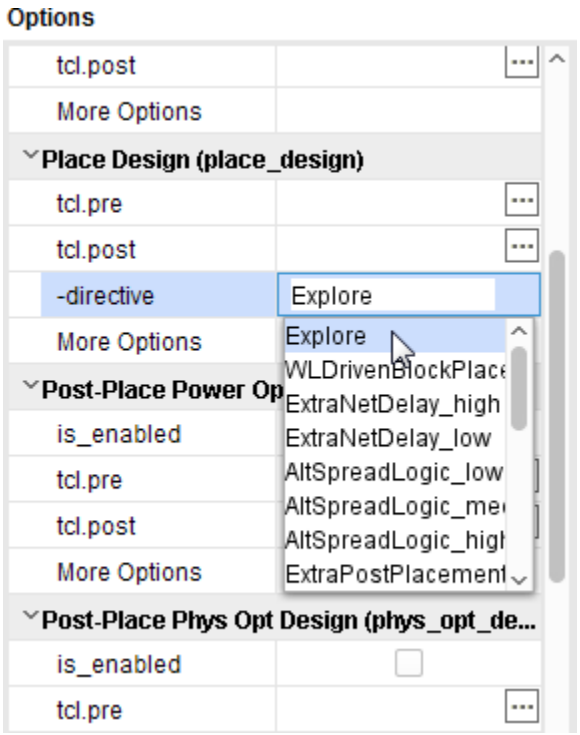
To review, copy, and modify run strategies:

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 the previous figure) 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 section Configurable Report Strategies in *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906).
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 create a copy of the currently selected strategy and add it to the User Defined Strategies list. Vivado then displays the strategy options on the right side of the dialog box for you to modify.
6. Provide a name and description for the new strategy as follows:
   - **Name:** Enter a strategy name to assign to a run.
   - **Type:** Specify Synthesis or 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.*



8. Click the right-side column of a specific option to modify command options. See the previous figure 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 *Vivado Design Suite User Guide: Using Tcl Scripting* (UG894).

     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. Here, `<InstallDir>` is the installation directory of the AMD software and `<version>` is the release version.

Launching Implementation Runs

You can launch the active implementation run, or select multiple runs to launch at once.

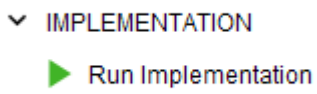
Launching a Single Implementation Run

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

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 **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 Multiple Runs

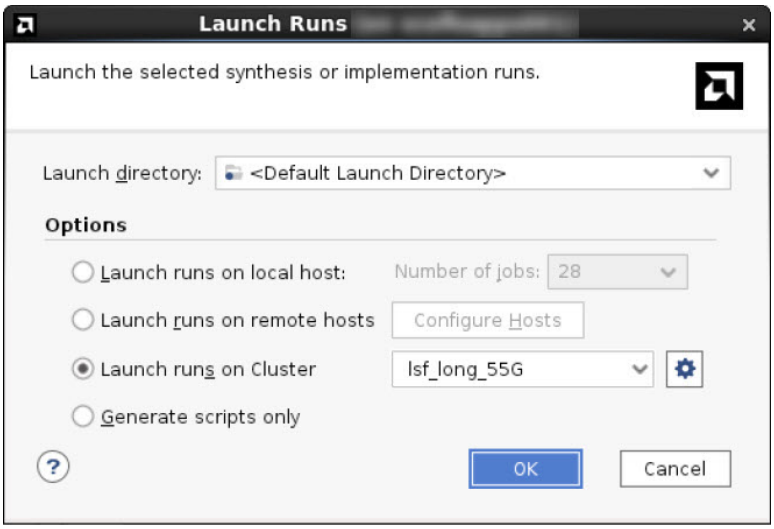
You can launch multiple runs at the same time by selecting them in the Design Runs window, as follows:

1. Use **Shift** or **Ctrl** 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 Runs dialog box, shown in the following figure.

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



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 **Launch runs on local host** 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) 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) 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. You can run the script 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 the following figure, 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! *Putting the process in the background blocks the Tcl Console. You cannot execute Tcl commands, or perform tasks that require Tcl commands, such as switching to another open design.*

Figure 8: **Starting Run - Background Process**

Running Implementation in Steps

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

- Opt Design (`opt_design`)

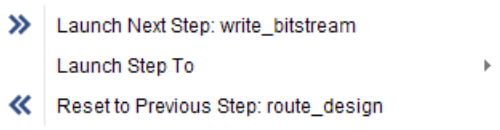
- Power Opt Design (`power_opt_design`) (optional)
- Place Design (`place_design`)
- Post-Place Power Opt Design (`power_opt_design`) (optional)
- Post-Place Phys Opt Design (`phys_opt_design`) (optional)
- Route Design (`route_design`)
- Post-Route Phys Opt Design (`phys_opt_design`) (optional)
- Write 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. See the following figure.



   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 fits it onto the target AMD 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 AMD 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 AMD device.
   - **Post-Route Phys Opt Design:** Optimizes logic, placement, and routing, using actual routed delays.
   - **Write Bitstream (all devices except Versal devices):** Generates a bitstream for AMD 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 AMD 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.

Take Non-Project based designs manually 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), or type `<command> -help`.

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

Implementation Sub-Processes

In project mode, the implementation commands are run in a fixed order. In non-project mode, you can run commands in a similar order repeatedly, iteratively, and in a different sequence than in the project mode.

IMPORTANT! *Implementation Commands are re-entrant.*

Implementation commands are re-entrant. When an implementation command runs in the non-project mode, Vivado reads the design in memory, performs its tasks, and writes the resulting design back into memory. This provides more flexibility when running in the 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 AMD 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 AMD device.
- **Power Opt Design (optional):** Optimizes design elements to reduce the power demands of the target AMD device.
- **Place Design:** Places the design onto the target AMD 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 AMD device.
- **Post-Route Phys Opt Design:** Optimizes logic, placement, and routing using actual routed delays (optional).
- **Write Bitstream:** Generates a bitstream for AMD 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.

This chapter documents each implementation step, its details, and the associated Tcl commands. The following table provides a list of sub-processes and their associated Tcl commands.

Table 4: **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).

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 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 file format is only supported for 7 series devices. 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, AMD 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 can 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 can be used to read the synthesized design into memory, depending on the source files in the design, and the state of the design. See the following table.

Table 5: **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 example of a synthesis run script.

```
# Setup design sources and constraints
read_vhdl -library bftLib [ glob ./Sources/hdl/bftLib/*.vhdl ]
read_vhdl ./Sources/hdl/bft.vhdl
read_verilog    [ glob ./Sources/hdl/*.v ]
read_xdc ./Sources/bft_full.xdc

# Run synthesis, report utilization and timing estimates, write design
checkpoint
synth_design -top bft -part xc7k70tfbg484-2 -flatten rebuilt
write_checkpoint -force $outputDir/post_synth
```

For more information on using the `synth_design` example script, see the *Vivado Design Suite Tutorial: Design Flows Overview* (UG888) and the *Vivado Design Suite User Guide: Synthesis* (UG901).

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 *Vivado Design Suite Tcl Command Reference Guide* (UG835). 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. You can use this command 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 section Understanding Project Mode and Non-Project Mode in the *Vivado Design Suite User Guide: Design Flows Overview* (UG892) for more information on Project Mode and Non-Project Mode.

IMPORTANT! *In the incremental compile flow, the `read_checkpoint` command is used to specify the reference design checkpoint, not the `open_checkpoint` command.*

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. Non-Project Mode does not support design runs.*

Use `open_run` before implementation on an RTL design 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.*

open_run Syntax

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

open_run Example Script

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

The `open_run` example script opens a design run (`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, Vivado tools prompt you to save changes before opening the new design.

RECOMMENDED: *After creating the in-memory synthesized design in the Vivado tools, review errors stop you from creating the in-memory synthesized design, 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 section Reading Design Modules in the *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909).

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 *Vivado Design Suite Tcl Command Reference Guide* (UG835).

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 devices. 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. The property must be used cautiously to avoid introducing too many global clocks into the design, which might result in placement failures. For more information, see the *Vivado Design Suite Properties Reference Guide* (UG912).

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.

Logic optimization also skips optimization of design objects that have physical constraints such as LOC, Bel, RLOC, LUTNM HLUTNM ASYNC_REG, and LOCK_PINS. 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. An optimization property applied to a design object such as a net or a cell forces that optimization to run, regardless of the specified options.

Table 6: **Optimization Ordering for Multiple Options**

Phase	Name	Option	Default
1	Retargeting	`-retarget`	X
2	Constant Propagation	`-propconst`	X
3	Sweep	`-sweep`	X
4[1]	Mux Optimization	`-muxf_remap`	
5[1]	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>`	
13	Control Set Optimization	`-control_set_opt`	
14	Remap	`-remap`	
15	Resynth Remap	`-resynth_remap`	

Table 6: **Optimization Ordering for Multiple Options** *(cont'd)*

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

Notes:

1. 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 7: **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
`-control_set_opt`	CONTROL_SET_OPT
`-muxf_remap`	MUXF_REMAP
`-propconst`	PROPCONST
`-remap`	REMAP
`-resynth_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 load-less cells and unconnected nets and does other optimizations, such as the following:

- Performs tie off on Macro Pins, flip-flop D pins
- Replicates flip-flops driving multiple OBUFs
- Based on feedback loop, updates with changes made to MMCM compensation attributes
- If only one output is used, retargets dual port RAMs to single port
- Optimizes SRLC32E/SRLC16E/ODDR/IDDR/CARRY4 cells for area (if possible)
- Removes unused IDELAYCTRL and groups IOs to be controlled by optimal number of IDELAYCTRL
- Inserts IBUFs and OBUFs to legalize connectivity for certain cells

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 AMD 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 in 7 series and UltraScale architectures and for Versal designs, remaps LOOKAHEAD8 primtives. 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.

TIP: *Remapping long carry chains to LUTs may significantly increase delay even with further optimization by adding the remap option. AMD 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 Multi-Clock Buffer (MBUFG) provides divide by 1, 2, 4, 8 clocks of the clock input on its O1, O2, O3, and O4 outputs. The MBUFG buffer routes the full-speed clock using a single global clock resource, and divides by 2, 4, or 8 as needed using local resources near the destination loads. MBUFG driven clocks consume fewer global routing resources. 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 replaces parallel clock buffers driven by a common driver or clock modifying block (CMB), such as MMCM, DPLL, or XPLL with a single MBUFG. The transformation occurs when the parallel clocks are derived from a common clock using divide ratios of 1, 2, 4, or 8, and when each clock buffer drives more than 50 loads. For CMB driven clocks, the phase shift has to be 0 and the duty cycle 50%. The MBUFG optimization is skipped when parallel buffers have conflicting clock constraints such as CLOCK_DELAY_GROUP or USER_CLOCK_ROOT, or when optimization would lose the original timing constraint intent. The following topologies are supported:

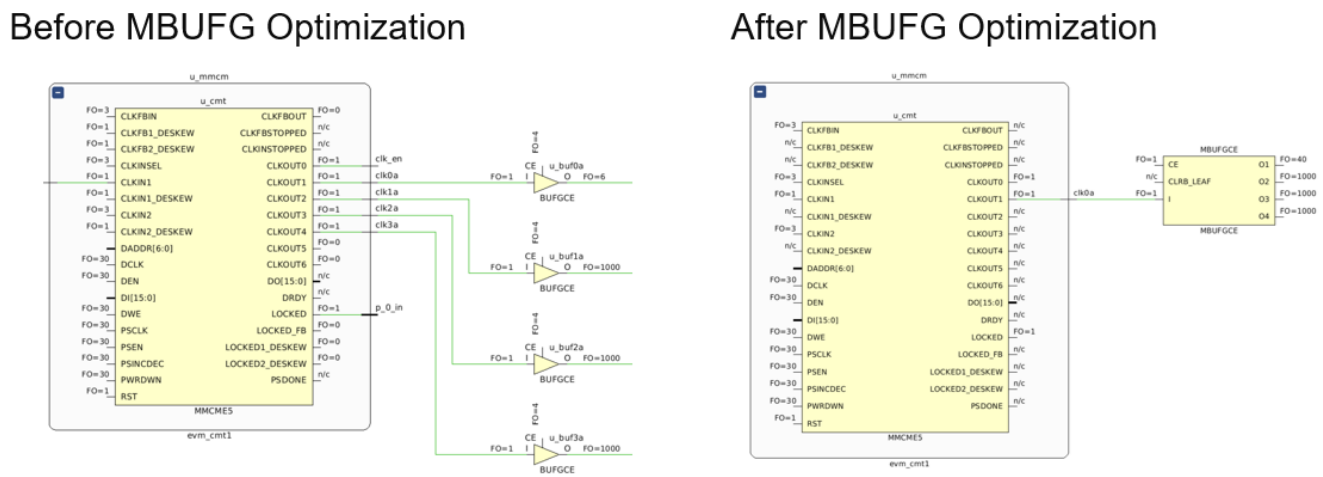
- Parallel BUFGCEs connected to a CMB are converted to an MBUFGCE.
- Parallel BUFGCE_DIVs connected to a common clock driver are converted to an MBUFGCE.
- Parallel BUFG_GTs connected to a common clock driver are converted to an MBUFG_GT.

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}]
```

The picture in the following figure shows an MMCM driving several BUFGCE buffers. The CLKOUTn driven clocks are integer divisions of 1, 2, 4, 8 of the CLKOUT1 driven clock. After the MBUFG optimization 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, and O4 pins.

Figure 9: **MBUFG Optimization**



Shift Register Optimization (Default)

Shift register (SRL) 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 can 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.

All transformations require that the SRL length is static and not controlled by an active signal. When a register is extracted from an SRL with a length of 1 (address=X"0"), the SRL is dissolved and converted to a register. This does not work only when it is driven by a cascade pin from another SRLC32E.

Transformations on cells using SRLC32E that use the cascade connection are restricted. However, operations are possible on the SRL chains. Extraction to input takes a delay from the final SRL and inserts registers at the input of the SRL chain when intermediate cells do not use the Q pin. Output registers are extracted from the final SRL only. All transforms from registers to SRLs require control sets to be 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. 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 converts from the overused 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 8: **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 0.5 ns), and timing to MREG is not critical (slack greater than 1 ns)
PREG to MREG	MREG=0, PREG=1	MREG=1, PREG=0	Timing to PREG is critical (slack less than 0.5 ns), and timing from PREG is not critical (slack greater than 1 ns).
MREG to ADREG	ADREG=0, MREG=1	ADREG=1, MREG=0	Timing to MREG is critical (slack less than 0.5 ns), and timing from MREG is not critical (slack greater than 1 ns)
ADREG to MREG	ADREG=1, MREG=0	ADREG=0, MREG=1	Timing from ADREG is critical (slack less than 0.5 ns), and timing to ADREG is not critical (slack greater than 1 ns)
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 0.5 ns), and timing from AREG/BREG is not critical (slack greater than 1 ns)
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 0.5 ns)
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 0.5 ns), and timing to AREG/BREG is not critical (slack greater than 1 ns)
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 0.5 ns), and timing from MREG is not critical (slack greater than 1 ns)
PREG push out to fabric	PREG=1	PREG=0, FDRE in fabric	Timing from PREG is critical (slack less than 0.5 ns), and timing to PREG is not critical (slack greater than 1 ns)
PREG pull in from fabric	PREG=0, FDRE in fabric	PREG=1	Timing from DSP output is critical (slack less than 0.5 ns)

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.

Control Set Optimization

This optimizes the design similarly to Control Set Reduction except that candidates are selected by the tool automatically.

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 can achieve further logic level reduction than Remap at the expense of longer compile time.

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.

opt_design lists the `opt_design` cell properties that trigger optimizations when using this option.

Resynth Remap

Remaps the design to improve the critical paths in timing-driven mode by performing re-synthesis to reduce the depth of logic. This timing-based approach replicates LUTs with fanout and collapse smaller LUTs into bigger functions at the expense of longer optimization compile time.

***Note*:** LUTs with BEL constraints are optimized by Resynth Remap. To prevent optimization on LUTs with BEL constraints, add a DONT_TOUCH property with value TRUE to the LUT.

opt_design

The `opt_design` command runs Logic Optimization.

opt_design Syntax

```
opt_design    [-retarget] [-propconst] [-sweep] [-bram_power_opt] [-remap]
        [-aggressive_remap] [-resynth_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] [-control_set_opt]
        [-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 ExploreArea
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.
- **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 compile time.
- **ExploreWithRemap:** Same as the Explore directive but includes the Remap optimization.
- **Default:** Runs opt_design with default settings.
- **RQS:** Instructs opt_design to select the directive specified by the report_qor_suggestion strategy suggestion. Requires an RQS file with a strategy suggestion to be read in prior to calling this directive.

The following table provides an overview of the optimization phase for the different directives.

Optimization Phases for Directives

Table 9: **Optimization Phases for Directives**

Default	Explore	ExploreArea	ExploreSequentialArea	ExploreWithRemap	RuntimeOptimized
Retarget	Retarget	Retarget	Retarget	Retarget	Retarget
Constant propagation	Constant propagation	Constant propagation	Constant propagation	Constant propagation	Constant propagation
Sweep	Sweep	Sweep	Sweep	Sweep	Sweep
BUFG optimization	BUFG optimization	BUFG optimization	BUFG optimization	BUFG optimization	BUFG optimization
Shift Register Optimization	Shift Register Optimization	Shift Register Optimization	Shift Register Optimization	Shift Register Optimization	Shift Register Optimization
Block RAM Power Opt[1]	Control Set Optimization[2]	Control Set Optimization[2]	Control Set Optimization[2]	Control Set Optimization[2]	
	MBUFG optimization[2]	Resynthesis	Resynthesis	Remap	
		MBUFG optimization[2]	MBUFG optimization[2]	MBUFG optimization[2]	

Notes:

1. Phase run in UltraScale/UltraScale+ designs.
2. Phase run in 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 section Synthesis Attributes in the *Vivado Design Suite User Guide: Synthesis* (UG901).

You 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 can 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 recommended approach for managing MARK_DEBUG usage is the `config_flows -mark_debug` option, which allows you to control optimization of objects with MARK_DEBUG without modifying source files or constraints. The following three values are supported:

- **enable:** Do not optimize MARK_DEBUG nets. This is a default value.
- **disable:** Allow both synthesis and implementation to freely optimize MARK_DEBUG nets.
- **synthesis_only:** Synthesis does not optimize MARK_DEBUG nets so that they are available at the beginning of implementation, but MARK_DEBUG nets can be freely optimized during implementation.

Property Based 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.

The following is a summary of properties for object-specific optimization.

Table 10: **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	Set to TRUE on SRL primitives to convert them to register chains
REG_TO_SRL	Set to TRUE on register chains to convert them to SRL primitives
SRL_STAGES_TO_REG_INPUT	Set to the appropriate value on an SRL primitive to move a register across its input
SRL_STAGES_TO_REG_OUTPUT	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
CLOCK_BUFFER_TYPE	Set on nets to insert corresponding Global Clock buffers
LUT_DECOMPOSE	Set on LUTs (LUT5, LUT6) for decomposition to reduce congestion

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 AMD 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).

***Note*:** Power Optimization is not supported for Versal designs.

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.

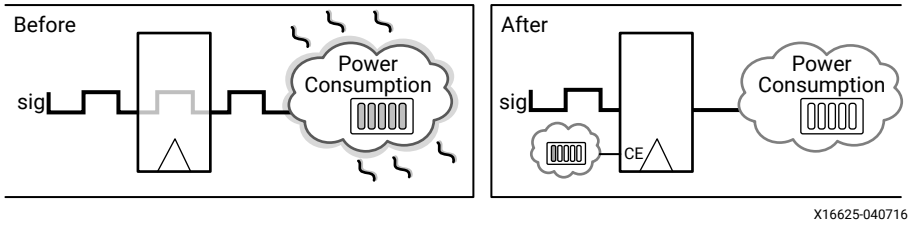
Using Clock Enables (CEs)

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 10: **Intelligent Clock Gating**



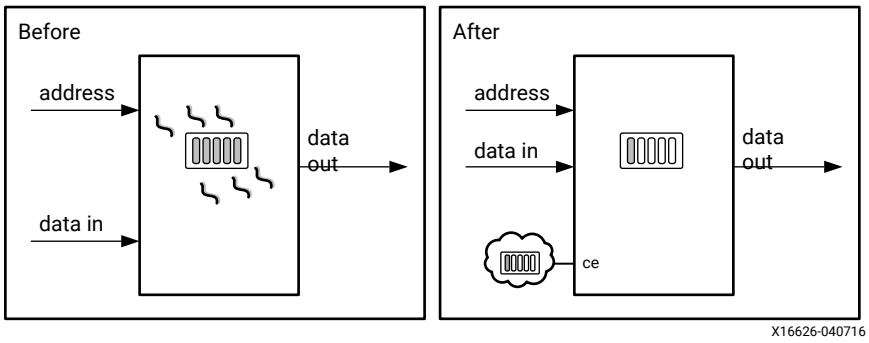
Intelligent clock gating also reduces power for dedicated block RAMs in either simple dual-port or true dual-port mode, as shown in the following figure.

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 11: **Leveraging Block RAM Enables**



***Note*:** Power Optimization is not supported for Versal using the Advanced Flow.

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>]`.*

***Note*:** power_opt_design and its related commands, set_power_opt and report_power_opt are not supported for Versal using the Advanced Flow.

Placement

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

Design Placement Optimization

The 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 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 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 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 AMD 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. It is done 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), *UltraScale Architecture Clocking Resources User Guide* (UG572) and *Versal Adaptive SoC Clocking Resources Architecture Manual* (AM003). For example, an input that drives a global clock buffer must be located at a clock-capable I/O site, in the same upper or lower half of the device for 7 series devices. The input must be located 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 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 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 can 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 *Versal Adaptive SoC Hardware, IP, and Platform Development Methodology Guide* (UG1387).

place_design for UltraScale and 7 Series

Placement Phases

The following section details the phases of placement after clock and IO placement is run for UltraScale and 7 series families. The phases are:

- Global Placement
- Detailed Placement
- 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. Pblock constraints are treated as hard during this phase, even if they have the IS_SOFT property set to True. 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 FPGAs and SoCs* (UG949).

Physical Synthesis Phase

During physical synthesis in the placer (PSIP), the placer performs physical optimizations to optimize the netlist for later placement phases based on the initial placement of the design after floorplanning. 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 design analysis 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 the following figure.

Figure 12: **Summary of Physical Synthesis Optimizations**

Summary of Physical Synthesis Optimizations

Optimization	WNS Gain (ns)	TNS Gain (ns)	Added Cells	Removed Cells	Optimized Cells/Nets	Dont Touch	Iterations	Elapsed
LUT Combining	0.000	1002.694	5302	31484	36786	0	1	00:03:15
Retime	0.000	0.000	0	0	0	0	1	00:00:00
Equivalent Driver Rewiring	0.000	21107.189	0	26	149	0	1	00:46:51
Very High Fanout	0.105	96521.977	2295	0	99	0	1	00:53:06
Fanout	0.000	13660.087	4728	0	600	0	1	03:58:38
Critical Cell	0.000	2046.885	224	0	363	0	1	00:01:15
DSP Register	0.000	0.000	0	0	0	0	1	00:00:02
Shift Register to Pipeline	0.000	0.000	0	0	0	0	1	00:00:01
Shift Register	0.000	248.757	292	0	160	0	1	00:00:43
BRAM Register	0.000	0.000	0	0	0	0	1	00:00:03
URAM Register	0.000	0.000	0	0	0	0	1	00:00:03
Dynamic/Static Region Interface Net Replication	0.000	0.000	0	0	0	0	1	00:00:00
Critical Cell	0.000	428.072	24	0	9	0	1	00:00:08
Total	0.105	135015.672	12865	31510	38166	0	13	05:44:05

- **Control Set Optimization:** Performs control set reduction with more accurate placement location information. This optimization only occurs with the Explore directive.
- **LUT Combining:** LUT combining is a placement optimization that maps two LUT primitives with common inputs to the same physical LUT site to reduce LUT utilization. LUTs are combined when non-timing-critical and un-combined when placement of the LUTs in separate locations can improve critical path delay.
- **Auto Pipeline Insertion:** Auto Pipeline Insertion is a non-timing driven optimization that inserts pipeline registers on user marked nets. This feature addresses timing closure challenges on specific buses and interfaces. Once pipeline stages have been inserted, the placer adjusts pipeline locations to improve clock speed. See Auto-Pipelining for more information.
- **Property-Based Retiming:**

  Property-based retiming provides user-controlled retiming through setting a property on a register or LUT. This optimization is ideal for critical paths with sufficient margin on timing startpoints or endpoints. Two properties control retiming in PSIP. PSIP_RETIMING_BACKWARD with value of TRUE performs backward retiming and PSIP_RETIMING_FORWARD with value of TRUE performs forward retiming. When PSIP_RETIMING_FORWARD with value TRUE is applied to a register, PSIP- forward retimes over all LUT loads driven by the Q pin of the register. When PSIP_RETIMING_FORWARD with value TRUE is applied to a LUT primitive, only the paths related to the LUT are impacted.

  When PSIP_RETIMING_BACKWARD with value TRUE is applied to a register, PSIP backward retimes over the LUT driving the D pin of the register. Note that the backward retiming property on the register does not trigger backward retiming over control set pin driver LUTs. When PSIP_RETIMING_BACKWARD with value TRUE is applied to a LUT primitive, the register driven by the LUT will be moved to LUT inputs.

  Multi-level retiming is supported by applying the property to all LUT primitives along the path. All retimed cells have the PHYS_OPT_MODIFIED property set to RETIMING.

  Retiming does not work for the following:

  - Moving logic across macro, such as BRAM, UltraRAM, and DSPs
  - Register packed in I/O sites
  - Paths with different startpoint/endpoint clocks
  - Paths with timing exceptions
  - Paths with properties that prevent optimization, such as DONT_TOUCH, ASYNC_REG, and so forth.
- **Very High-Fanout Optimization:** Very High-Fanout Optimization replicates registers driving high-fanout nets (fanout > 1000, slack < 2.0 ns).

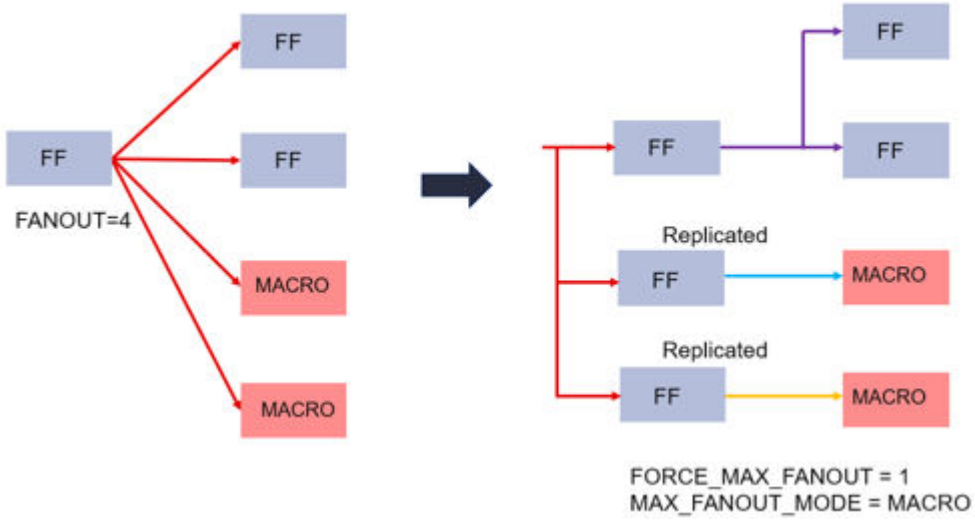
- **Critical Cell Optimization:** Critical-Cell Optimization replicates cells in failing paths. If the loads on a specific cell are placed far apart, the cell can be replicated with new drivers placed closer to load clusters. This optimization often applies to nets driving large block RAM or UltraRAM arrays. It also applies to a 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 is 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 is clustered together.

  The MAX_FANOUT_MODE property takes precedence over the FORCE_MAX_FANOUT property. Physical synthesis tries to honor both by applying MAX_FANOUT_MODE based optimization first and then all its replicated drivers inherit the FORCE_MAX_FANOUT property to do further replication within a clock region. This is illustrated in the following figure 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 13: **Applying MAX_FANOUT_MODE with value MACRO together with FORCE_MAX_FANOUT**



Note: This optimization happens early in the placer. In the later stages as the timing accuracy improves, both the replicated source and/or load registers can be moved to different clock regions or SLRs if the timing estimate improves.

- **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.

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_MODIFIED and PHYS_OPT_SKIPPED values that correspond with a Physical Synthesis optimization.

Table 11: **Physical Synthesis Optimization**

Optimization	PHYS_OPT_MODIFIED and PHYS_OPT_SKIPPED Value
Autopipeline Insertion	AUTOPIPELINE
Block RAM Register Optimization	BRAM_REGISTER_OPT
Control Set Optimization	CONTROL_SET_OPT
Critical Cell optimization	CRITICAL_CELL_OPT
DSP Register Optimization	DSP_REGISTER_OPT
Equivalent Driver Rewire Optimization	EQU_REWIRE_OPT
Fanout Optimization	FANOUT_OPT
Negative-edge Triggered Register Insertion	INSERT_NEGEDGE
Placement Optimization	PLACEMENT_OPT
Property Based Retiming	RETIMING
Shift Register Optimization	SHIFT_REGISTER_OPT
Shift Register to Pipeline Optimization	SHIFT_REGISTER_TO_PIPELINE
SLR Crossing Optimization	SLR_CROSSING_OPT
URAM Register Optimization	URAM_REGISTER_OPT
Very High Fanout Optimization	FANOUT_OPT

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, block RAM, 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, BUFG Replication, and the optional BUFG insertion phase. In the BUFG Replication phase, BUFG driven nets that span multiple SLRs will receive their own BUFG driver for each SLR for non-Versal devices. The optimization is skipped in case of placement or routing conflicts, constraints that would prevent replication, or timing degradation. In the BUFG insertion phase, the placer can route high fanout nets on global routing tracks to free up fabric routing resources. High-fanout nets with a (fanout > 1,000 for UltraScale and UltraScale+ and fanout > 10,000 for Versal) driving control signals with a slack greater than 1.0 ns 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. The optimization is also skipped if netlist editing required by the optimization fails. 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 logic restructuring, physical optimization, or floorplanning to achieve timing closure.*

place_design Command (7 Series and UltraScale)

***Note*:** This section applies to 7 series and UltraScale families only. Refer to place_design Command (Versal) for Versal families.

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 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] [-sll_align_opt]
              [-no_bufg_opt] [-ultrathreads] [-quiet] [-verbose]
```

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_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. The following table shows which directives might benefit which types of designs.

Table 12: **Directive Guidelines**

Directive	Designs Benefited
`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.
- **RQS:** Instructs place_design to select the directive specified by the report_qor_suggestion strategy suggestion. Requires an RQS file with a strategy suggestion to be read in prior to calling this directive.

Auto Directives

When closing timing on challenging designs, users might choose to run many different place_design directives to select the best timing result. Auto directives use machine learning to predict the best directives to run. Users can benefit by only running these directives instead of the full sweep of directives listed in Available Directives.

Note: When running with the Auto directives, the directive setup happens slightly later in the flow than when the directive is directly specified which can result is slightly different results.

Machine learning prediction of directives have a margin of error. As a consequence, it is recommended to run 3 *Auto_n* directives and take the best result. The directives predicted are equivalent to the ones mentioned in Available Directives, so there is no benefit in running auto directives in addition to these. The directive selected by the tool is reported in the log file. An example of the message is as follows:

```
INFO: [Place 30-1947] Predicted directive using ML models is:
EarlyBlockPlacement
```

To enable the feature, set the `place_design -directive <value>` where value is:

- **Auto_1:** High performing predicted directive
- **Auto_2:** Second best predicted directive
- **Auto_3:** Third best predicted directive

Switches

Using the -no_bufg_opt Option

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

Using the -no_psip Option

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

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 -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 -sll_align_opt Option

The `-sll_align_opt` option attempts to align registers driving super long lines in SSI multi die parts. It is ignored for monotlithic parts.

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 -ultrathreads option

This option is available only for UltraScale+ SSI and vu440 devices and speeds up placement by distributing multiple threads indicated by `general.maxThreads` as evenly as possible across multiple SLRs. Placement results will differ slightly depending on whether or not ultrathreads is used.

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 -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.

Examples

In the following example, 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.

```
# 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
```

place_design for Versal

Placement Phases

Floorplanning

Floorplanning phase consists of partitioning, BUFG_FABRIC replication, NOC placement, clock region assignment/placement, and Physical Synthesis in Placer (PSIP). Floorplanning algorithm is revised in the Advanced Flow.

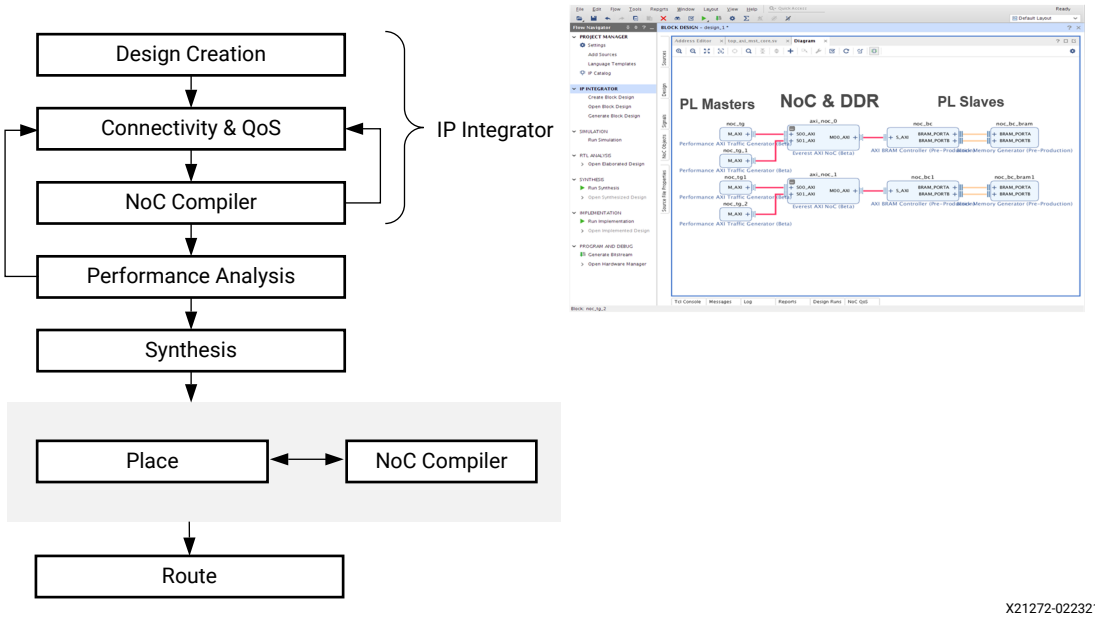
BUFG_FABRIC Replication for HFNs

For Versal designs, there is a dedicated buffer to use the clock tree network for non-clock pins called BUFG_FABRIC. BUFG_FABRIC cells are inserted during opt_design to drive high fanout nets (HFN) with more than 25,000 logical loads. For SSI designs, if loads of a BUFG_FABRIC global buffer span multiple SLRs, the BUFG_FABRIC is replicated in each SLR so that each replica only drives loads within a single SLR. This reduces the overall high-fanout net delay. This replication occurs before clock placement to ensure that each BUFG_FABRIC resource is counted against the available global clock buffer resources, thereby avoiding global clock buffer overutilization in placement. Depending on the clock utilization, these cells are replicated once again towards the end of placer during the BUFG_FABRIC Optimization phase to driver loads located in a single common clocking column partition of each SLR leading to further QoR enhancements. Furthermore, nets that were not previously promoted to BUFG_FABRIC by opt_design are opportunistically targeted by placer.

NoC Compiler Runs During Placement

The Vivado IP integrator invokes the NoC Compiler during block design validation to generate a NoC placement and routing solution to meet Quality of Service (QoS) requirements. If the solution from IP integrator does not sufficiently meet design implementation requirements, the NoC Compiler might be invoked during design placement to generate a new solution to meet the implementation requirements.

Figure 14: **NoC Compiler Flow**



Following are implementation requirements that might cause the NoC Compiler to be invoked during design placement:

- Physical location or Pblock constraints applied to the Programmable Logic (PL) that influences NoC NoC Master Unit (NMU)/ NoC Slave Unit (NSU) placement
- Resolution of the NoC interface between CIPS and NoC for proper assignment to the targeted device
- Top-level port assignment of DDR memory controller interfaces that results in a change in DDR memory controller assignment
- Global placement of programmable logic that would influence NoC NMU/NSU placement

> **TIP:** *In the IP integrator, you can constrain the location of the DDR memory controller to the appropriate site in the NoC View to reflect the assignment to perform during design placement. This improves the NoC QoS results correlation between IP integrator and a fully implemented design.*

- The NOC compiler runs in the preplace mode, so the placement of fabric is driven by the placement of NoC instances that result in better NoC QoS.

Global Placement

The global placement algorithm in Advanced Flow models objectives such as wirelength, congestion, and timing more accurately. This new approach delivers better results with reduced wirelength and congestion overhead. Additionally, a new congestion optimization engine effectively alleviates long congestion, which was a challenge for the previous global placer.

Physical Synthesis Phase

During physical synthesis in the placer (PSIP), 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 the following figure.

Figure 15: **Summary of Physical Synthesis Optimizations**

Summary of Physical Synthesis Optimizations

Optimization	WNS Gain (ns)	TNS Gain (ns)	Added Cells	Removed Cells	Optimized Cells/Nets	Dont Touch	Iterations	Elapsed
SLR Replication	0.000	5042.058	156	0	156	0	1	00:00:19
Very High Fanout	0.000	2946.884	1378	0	56	0	1	00:00:53
Retime	0.000	0.000	0	0	0	0	1	00:00:00
Equivalent Driver Rewiring	0.000	3722.454	0	755	150	0	1	00:02:04
Post Floorplan Control Set	0.000	0.002	570	0	570	0	1	00:00:11
Fanout	0.125	491.634	7	0	52	0	1	00:00:08
Critical Cell	0.000	-8.762	7	0	2	0	1	00:00:00
DSP Register	0.000	0.000	0	0	0	0	1	00:00:01
BRAM Register	0.000	0.000	0	0	0	0	1	00:00:02
URAM Register	0.000	0.000	0	0	0	0	1	00:00:03
Dynamic/Static Region Interface Net Replication	0.000	0.000	0	0	0	0	1	00:00:00
Critical Cell	0.000	0.000	0	0	0	0	1	00:00:00
Total	0.125	12194.270	2118	755	986	0	12	00:03:41

- **SLR Replication:**

  High fanout nets that connect a single driver to multiple loads, can present timing challenges when these loads are distributed across different SLRs. SLR replication replicates the driver of high fanout nets which has critical loads in other SLRs. Drivers are selected based on timing estimates after Floorplanning phase of placer. In this optimization all soft and hard Pblock constraints are honored.

- **Control Set Optimization:** Performs control set reduction with more accurate placement location information. With meaningful initial placement result, the flops are distributed among the placement area, and flop minimal resource usage for the exactly flops in a small region is calculated. For the hotspot regions, it finds an optimal solution to reduce the resource usage. Therefore, reduce the legalization effort to push cells further in downstream and leave more room for other optimizations. This phase is activated during placer's Explore directive only.

- **Auto Pipeline Insertion:** Auto Pipeline Insertion is a non-timing driven optimization that inserts pipeline registers on user marked nets. This feature is used to address timing closure challenges on specific buses and interfaces. Once pipeline stages have been inserted, the placer adjusts pipeline locations to improve clock speed. See Auto-Pipelining for more information.

- **Property-Based Retiming:** Property-based retiming provides user-controlled retiming through setting a property on a register or LUT. This optimization is ideal for critical paths with sufficient margin on timing startpoints or endpoints. Two properties control retiming in PSIP. PSIP_RETIMING_BACKWARD with value of TRUE performs backward retiming and PSIP_RETIMING_FORWARD with value of TRUE performs forward retiming. The properties

can be applied to a register or LUT. When PSIP_RETIMING_FORWARD with value TRUE is applied to a register, PSIP forward- retimes over all LUT loads driven by the Q pin of the register. When PSIP_RETIMING_FORWARD with value TRUE is applied to a LUT primitive, the register driving the LUT is moved to the output of the LUT. When PSIP_RETIMING_BACKWARD with value TRUE is applied to a register, PSIP backward retimes over the LUT driving the D pin of the register. Note that the backward retiming property on the register does not trigger backward retiming over control set pin driver LUTs. When PSIP_RETIMING_BACKWARD with value TRUE is applied to a LUT primitive, the register driven by the LUT will be moved to LUT inputs. Multi-level retiming is supported by applying the property to all LUT primitives along the path. All retimed cells will have the PHYS_OPT_MODIFIED property set to RETIMING.

Retiming does not work for the following:

- Moving logic across macro, such as BRAM, UltraRAM, and DSPs
- Register packed in I/O sites
- Paths with different startpoint/endpoint clocks
- Paths with timing exceptions
- Paths with properties that prevent optimization, such as DONT_TOUCH, ASYNC_REG, and so forth.

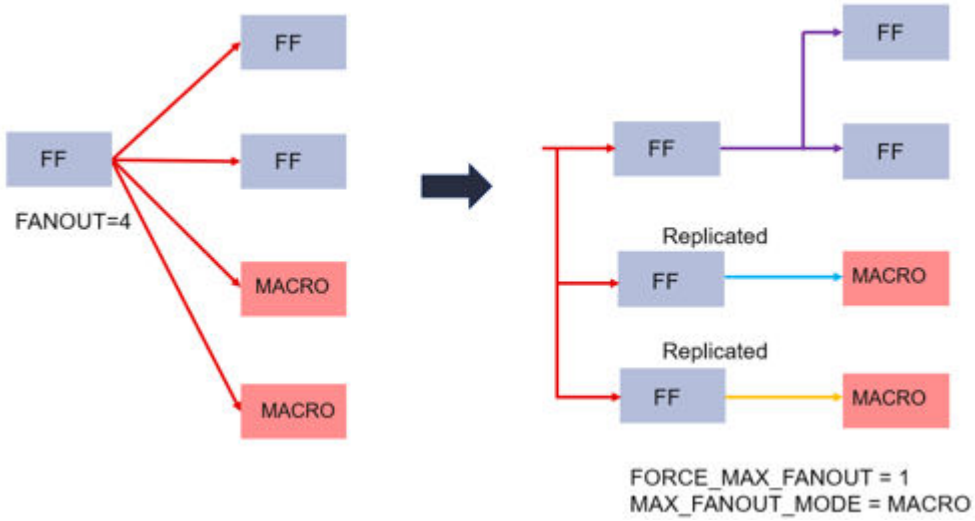
- **Very High-Fanout Optimization:** Very High-Fanout Optimization replicates registers driving high-fanout nets (fanout > 1000, slack < 2.0 ns).

- **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 block RAM 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 following figure 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 16: **Applying MAX_FANOUT_MODE with value MACRO together with FORCE_MAX_FANOUT**



***Note*:** This optimization happens early in the placer. In the later stages of the placer as the timing accuracy improves, both the replicated source and/or load registers may be moved to different clock regions or SLRs if the timing estimate improves.

- **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 greater than 1 to a dynamically adjusted register pipeline. It pulls more registers when the distance to cover is longer. The registers are placed optimally to balance timing paths. The latency is unaffected.

  Only SRLs with the PHYS_SRL2PIPELINE attribute set to TRUE are considered for this optimization. The property must be set on the SRL cell and should be set on all bits of a bus to have all the bits optimized. The pull of FFs happens on the SRL's Q-pin.
- **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.
- **Physical Optimization of SLR crossings:** The physical optimization of SLR crossing pulls out registers from the SRL if there is an SLR crossing. It guides the tool to have an optimal FF->FF SLR crossing to increase the performance on the SLR crossing by using the USER_SLL_REG=TRUE property. If, after the optimization, the SRL depth becomes one, it is converted into a regular register.

  The optimization occurs when:

  - Fanout of the SRL is 1
  - Minimum clock frequency threshold is 250 MHz
  - SRL needs to have a minimum depth of 2
  - SRL is not LUT combined
  - Source and destination cells are in neighboring SLRs

  Supported circuits:

  1. SRL → SLR Crossing → BRAM/URAM/DSP
  2. SRL → FF → SLR Crossing → BRAM/URAM/DSP
  3. SRL → SLR Crossing → FF → BRAM/URAM/DSP
  4. BRAM/URAM/DSP→ SLR Crossing → SRL
  5. BRAM/URAM/DSP→ FF → SLR Crossing → SRL
  6. BRAM/URAM/DSP→ SLR Crossing → FF → SRL
  7. LUT→ SLR Crossing → SRL
  8. SRL→ SLR Crossing → LUT

  Example:

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.

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_MODIFIED and PHYS_OPT_SKIPPED values that correspond with a Physical Synthesis optimization.

Table 13: **Physical Synthesis Optimization**

Optimization	PHYS_OPT_MODIFIED and PHYS_OPT_SKIPPED Value
Autopipeline Insertion	AUTOPIPELINE
Block RAM Register Optimization	BRAM_REGISTER_OPT
Control Set Optimization	CONTROL_SET_OPT
Critical Cell optimization	CRITICAL_CELL_OPT
DSP Register Optimization	DSP_REGISTER_OPT
Equivalent Driver Rewire Optimization	EQU_REWIRE_OPT
Fanout Optimization	FANOUT_OPT
Property Based Retiming	RETIMING
Shift Register Optimization	SHIFT_REGISTER_OPT
Shift Register to Pipeline Optimization	SHIFT_REGISTER_TO_PIPELINE
URAM Register Optimization	URAM_REGISTER_OPT
Very High Fanout Optimization	FANOUT_OPT

Detailed Placement

The Detailed Placement (DP) in Advanced flow is more robust compared to the non-Versal flow. It handles nested and overlapping Pblocks efficiently. If the packing solution is feasible, the legalization guarantees a solution.

Key benefits of Detailed Placement in Advanced Flow include:

- Full parallelization for significant speedup.
- Improved support for USER_SLL_REG, especially for multiple fanout and DFX designs.
- Support for the IS_DUT constraint for readback applications.

place_design Command (Versal)

***Note*:** This section refers only to the place_design command for Versal families. For other families, refer to place_design Command (7 Series and UltraScale).

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 placer uses the existing placement as the starting point instead of starting from scratch.

place_design Syntax

```
place_design  [-directive <arg>] [-subdirective <args>] [-no_timing_driven]
              [-timing_summary] [-unplace] [-no_psip] [-no_noc_opt]
              [-clock_vtree_type <arg>] [-net_delay_weight <arg>]
              [-quiet] [-verbose]
```

Table 14: **PSIP Optimizations**

PSIP Optimization	Applicable on which objects
Critical cell	Nets
Fanout	Nets
Very high fanout	Nets
Equivalent Driver Rewire	Nets
DSP Register	Cells
Block RAM Register	Cells
URAM Register	Cells
Shift Register	Cells
Logic Retiming	Cells
Per SLR Replication	Nets
Dynamic/Static Region Interface Net replication	Nets
Shift Register to Pipeline	Cells
Auto Pipeline Insertion	Nets
Control Set optimization	Cells

Using Directives and Sub-Directives

Directives

Directives provide different modes of behavior for the `place_design` command. They control the overall placement strategy. Only one directive can be specified at a time. The directive option is incompatible with other options with the exception of `-subdirective`, `-clock_vtree_type`, `-no_psip`, `-quiet`, and `-verbose`.

The `-directive <arg>` option can be set as follows:

- **Explore:** Increased placer effort in detail placement and post-placement optimization.
- **AggressiveExplore:** Attempt to improve QoR at the expense of compile time. The placer compile time might be significantly higher compared to the Explore directive because the placer uses more aggressive optimization goals to attempt to meet timing requirements.

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

Sub-Directives

Sub-directives allow control over the placer phases. More than one sub-directive might be used at a time. The `-subdirective` option is incompatible with other options with the exception of `-directive`, `-no_psip`, `-clock_vtree_type`, `-quiet` and `-verbose`.

The `-subdirective` option has the following format:

- **For sub-directives that must be set to a value (typically low|med|high):** <phase>.<sub-directive>.<value>
- **For sub-directives with no value:** <phase>.<sub-directive>

There are three possible values of phase that correspond to the phases of the placer that accept sub-directives. These are:

- **Floorplan:** For the floorplanning phase
- **GPlace:** For the global placer phase
- **DPlace:** For the detailed placer phase

Supported sub-directives are:

- **RuntimeOptimized:** Improves compile time at the expense of design performance.
  - Supported phases: Floorplan, GPlace, DPlace
  - Values: Using option indicates optimization is enabled
- **ExtraTimingUpdate:** Increases the number of timing updates to improve design performance.
  - Supported phases: Floorplan, GPlace, DPlace
  - Values: Using option indicates optimization is enabled
- **ExtraTimingOpt:** Improves design performance at the expense of compile time.
  - Supported phases: Floorplan, GPlace, DPlace
  - Values: low (default), med, high
- **BalancedSLR:** Partitions across SLRs to balance number of cells between SLRs.
  - Supported phases: Floorplan

  - Values: low, med (default), high
- **ForceSpreading:** Spreads the placement by lowering the target utilization for all block types
  - Supported phases: Floorplan, GPlace
  - Values: low (default), med, high
- **ReduceCongestion:** Reduces global and pin congestion at the expense of other metrics.
  - Supported phases: GPlace
  - Values: low (default), med, high
- **WLDrivenBlockPlacement:** Wirelength-driven placement of RAM and DSP blocks. Overrides 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.
  - Supported phases: Floorplan, GPlace
  - Values: Using option indicates optimization is enabled
- **ReducePinDensity:** Detail placer tries to reduce pin congestion for CLBs to improve overall router convergence.
  - Supported phases: DPlace
  - Values: low (default), med, high
- **EarlyBlockPlacement:** Places RAM and DSP blocks early in global placement and uses them as anchors for placing remaining logic.
  - Supported phase: GPlace

Switches

The following switches to `place_design` are applicable to the Versal family only.

Using the -clock_vtree_type Option

The `-clock_vtree_type` option is used in `place_design` to specify the type of clock tree to be used. The valid values are balanced, interSLR, and intraSLR. The default value is balanced. This option does not affect clocks using calibrated deskew.

Use the `-clock_vtree_type` option to select the clock tree that minimizes the clock skew for the types of timing challenges seen in the design:

- Select **intraSLR** to minimize clock skew within each SLR
- Select **interSLR** to minimize clock skew on paths between SLRs
- Select **balanced** for the best compromise between intraSLR and interSLR

***Note*:** Placer clock V-tree type properties are case sensitive. Using the wrong case generates an error message stopping the placer flow. Accepted types are: balanced, interSLR, intraSLR.

Using the -net_delay_weight Option

The `-net_delay_weight` option impacts how the placer sees net delays. Higher net delays force the placer to place cells closer together to meet the timing goal. Timing might improve but might result in increased congestion. Valid options are low, medium, and high.

- **Default:** low

Using the -no_noc_opt Option

The NoC compiler runs in the preplace mode, so the placement of fabric is driven by the placement of NoC instances, which results in better NoC QoS.

Using the -no_psip Option

The `-no_psip` option disables the physical synthesis algorithm in the placer.

Using the -psip_options Option

The -psip_options command line option is used in place_design to specify a list of optimizations to switch on during Physical Synthesis Phase. This switch can be used to exclude any specific optimization that is being triggered. Optimizatons are only performed if applicable for the given design.

You can specify different options using the following command:

```
place_design -psip_options {opt1 opt2}
```

For example, if you want to run only property-based retiming and fanout optimization during the Physical Synthesis Phase, you can run:

```
place_design -psip_options {retime fanout_opt}
```

Table 15:

Optimization	Opt name
Critical Cell Optimization	critical_cell_opt
Fanout Optimization	fanout_opt
Very High Fanout Optimization	very_high_fanout_opt
Equivalent Driver Rewire Optimization	equ_drivers_opt
DSP Register Optimization	dsp_register_opt
Block RAM Register Optimization	bram_register_opt
URAM Register Optimization	uram_register_opt
Shift Register Optimization	shift_register_opt
Property-Based Retiming	retime

Table 15: (cont'd)

Optimization	Opt name
Per-SLR Replication Optimization	per_slr_replication_opt
Shift Register to Pipeline Optimization	shift_register_to_pipeline
Auto Pipeline Insertion Optimization	auto_pipeline_insertion
Control Set Optimization	control_set_opt

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 compile 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 -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 -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 find it helpful.

Examples

The following example directs the placer to try different placement algorithms to achieve a better placement result:

```
place_design -directive Explore
```

The following example uses the Default directive but enables wirelength driven placement of RAM and DSP blocks during floorplanning.

```
place_design -subdirective Floorplan.WLDrivenBlockPlacement
```

The following example uses the Default directive but instructs the placer to use high effort for SLR balancing in the floorplanning phase, reducing congestion in global place and reducing pin density in the detailed placement phase.

```
place_design -subdirective {Floorplan.BalancedSLR.high
GPlace.ReduceCongestion.high DPlace.ReducePinDensity.high}
```

LUT Combining in Advanced Flow (Versal) Placement

In Advanced Flow (Versal) placement, LUT combining is done differently compared to the non-Versal Vivado placement. The Advanced Flow uses a more powerful packing engine, which performs more aggressive LUT combining throughout the placement flow as well as it has the capability to break it subsequently if any combined LUTs are on critical paths. Unlike non-Versal flow, where there is an explicit step for LUT combining during PSIP through LUTNM shape creation, the Advanced Flow placer does not have a specific LUT combining step. Instead, LUT combining can occur during the different phases of the placement flow. In non-Versal Vivado flow, the large portion of LUT combining is done during PSIP along with some opportunistic LUT combining in the flow and the users have an option to turn off PSIP part of the functionality. However, in the Advanced Flow, there is no option available to turn off LUT combining.

In summary, the Advanced Flow's packing engine inherently handles LUT combining more aggressively, eliminating the need for an explicit LUT combining step as seen in non-Versal flow.

***Note*:** LUT Combining does not alter the netlist. It uses two physical LUTs within a single LUT BEL.

***Note*:** After the design has been placed, the following table generated from `report_utilization` reports dual output LUTs. Dual output LUTs are made up of combined LUTs, LUT6CY, and LUT6_2s. From the Primitive table, subtract the number of LUTCY1s and LUT6_2s from the number to get a LUT combined figure.

Figure 17: **CLB Distribution**

2. CLB Distribution

Site Type	Used	Fixed	Prohibited	Available	Util%
SLICE	1021801	0	0	1057536	96.62
SLICEL	512895	0			
SLICEM	508906	0			
using Distributed RAM or Shift Registers	3111	0			
CLB LUTs	5194747	0	0	8460288	61.40
using CASCADE	74319	0			
LUT as Logic	5173596	0	0	8460288	61.15
single output	4741585				
dual output	432011				
LUT as Memory	21151	0	0	4230144	0.50
LUT as Distributed RAM	16200	0			
single output	2744				
dual output	13456				
LUT as Shift Register	4951	0			
single output	3580				
dual output	1371				
CLB Registers	2233390	0	0	16920576	13.20
Register driven from within the CLB	1365237				
Register driven from outside the CLB	868153				
LUT in front of the register is unused	368382				
LUT in front of the register is used	499771				
CLB Imux registers	0	0			
Pipelining	0				
Unique Control Sets	88830		0	2115072	4.20

Auto-Pipelining

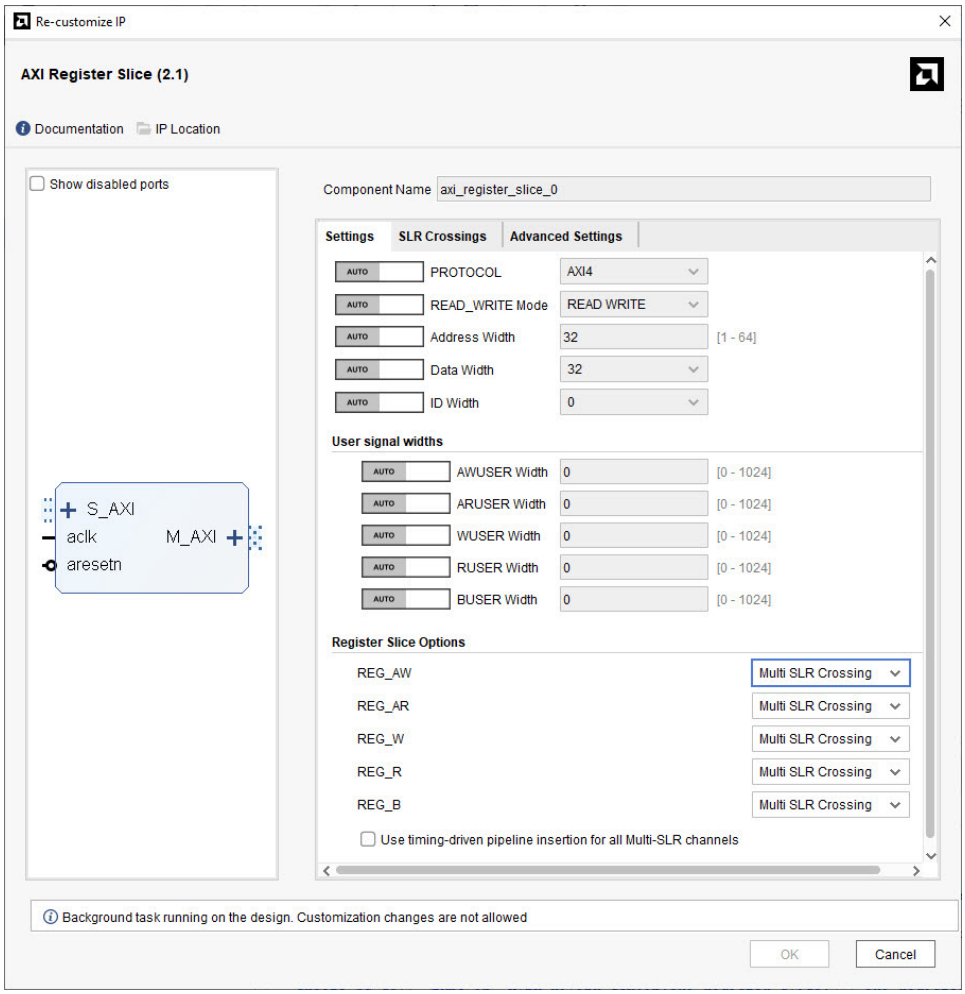
You can optionally insert additional pipeline registers during placement to address timing closure challenges on specific buses and interfaces.

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 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 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).

Table 16: **Properties for Auto-Pipelining on Custom Interfaces**

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.

Table 16: **Properties for Auto-Pipelining on Custom Interfaces** *(cont'd)*

Property Name	Object	Format/Range	Description
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, AMD 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 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.

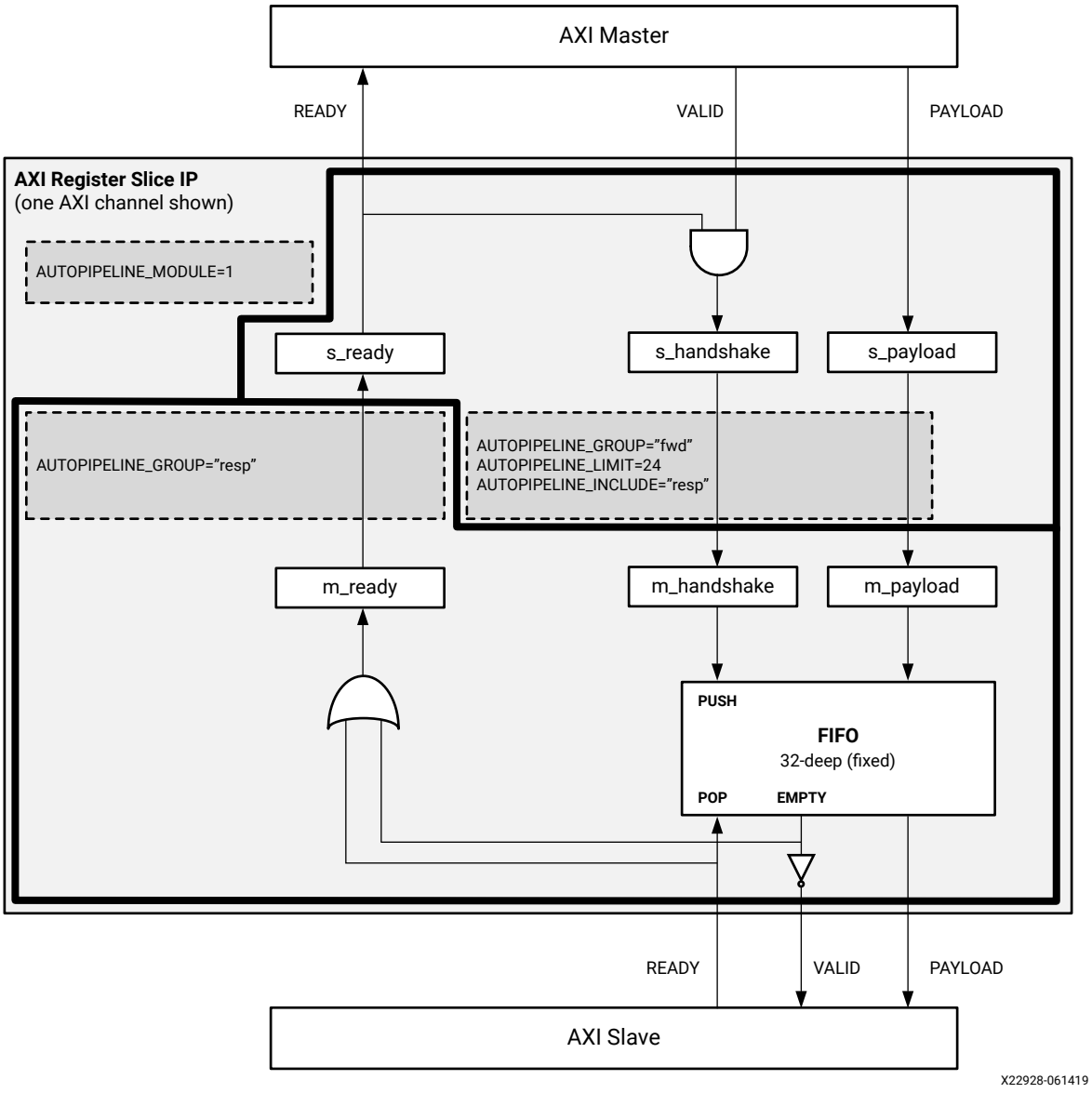
The following figure shows how the auto-pipeline properties are used in the AXI Register Slice RTL.

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

```
(* 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;
```

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

Figure 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.

Figure 21: **Example of 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           |
```

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

Figure 22: **Summary of Physical Synthesis Options for Auto Pipeline 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 |
-------------------------------------------------------------------------------------------------------------------
```

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

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

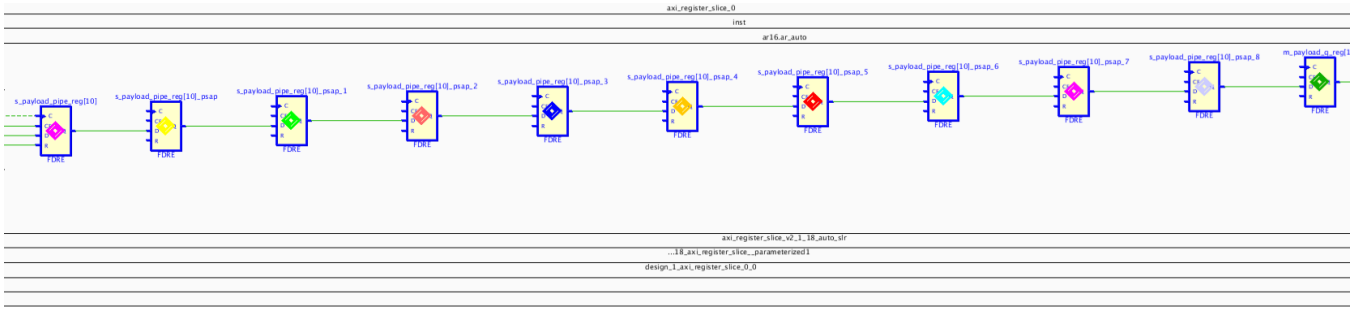
Figure 23: **Summary of Physical Synthesis Optimizations for Pipeline2SRL Table**

Summary of Physical Synthesis Optimizations

Optimization	WNS Gain (ns)	TNS Gain (ns)	Added Cells	Removed Cells	Optimized Cells/Nets	Dont Touch	Iterations	Elapsed
Pipeline to Shift Register	0.000	10.695	5042	21064	5042	0	1	00:07:59
Total	0.000	10.695	5042	21064	5042	0	1	00:07:59

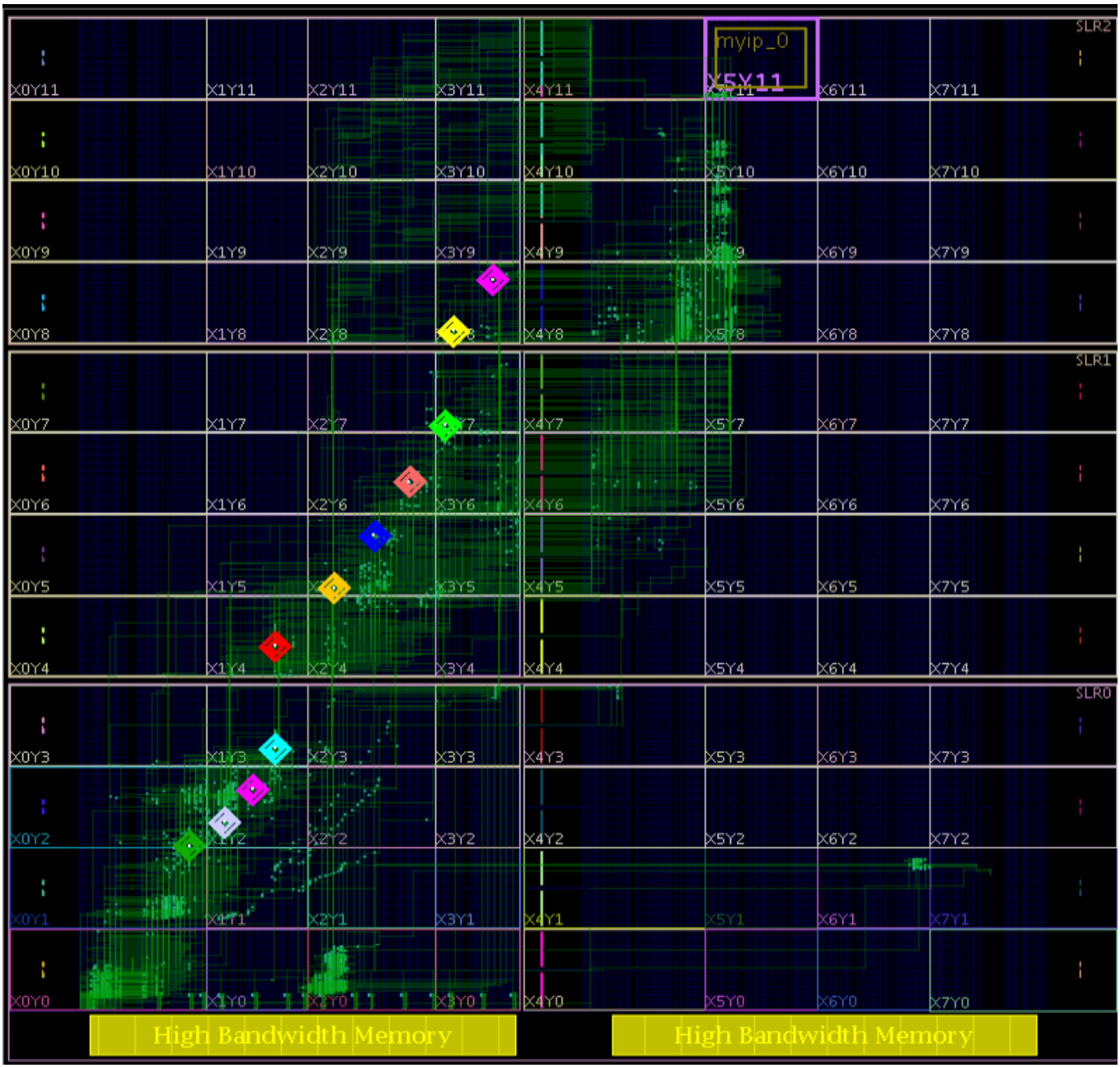
The following figure shows the path from SLR2 to SLR0 where nine pipeline stages were automatically inserted during `place_design`.

Figure 24: **Schematic View of Auto-Pipeline Inserted Registers**



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

Figure 25: **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 compile time 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 17: **Post-Place and Post-Route Physical Optimizations**

Option Name	post-place		post-route	
	valid	default	valid	default
Critical Cell Optimization	Y[1]	Y[1]	Y[1]	N
Fanout Optimization	Y[1]	Y[1]	N	N/A
Very High Fanout Optimization	Y[1]	Y[1]	N	N/A
Interconnect Retiming	Y[2]	Y[2]	Y[2]	Y[2]
Critical Cell Group Optimization	Y[2]	Y[2]	Y[2]	N

Table 17: **Post-Place and Post-Route Physical Optimizations** *(cont'd)*

Option Name	post-place		post-route	
	valid	default	valid	default
Clock Optimization	Y[2]	Y[2]	Y	Y
DSP Register Optimization	Y	Y	N	N/A
Block RAM Register Optimization	Y	Y	N	N/A
URAM Register Optimization	Y	Y	N	N/A
Shift Register Optimization	Y	Y	N	N/A
Critical Pin Optimization	Y	Y	Y	Y
LUT Restructure Optimization	Y	Y	Y	N
Single LUT Optimization	Y[2]	Y[2]	Y[2]	Y[2]
LUT Cascade Optimization	Y[2]	Y[2]	Y[2]	N
Placement Optimization	Y[1]	Y[1]	Y[1]	Y[1]
Routing Optimization	N	N/A	Y	Y
Block RAM Enable Optimization	Y[1]	N	N	N/A
Hold-Fixing	Y	N	Y	N
Negative-Edge FF Insertion	Y[1]	N	N	N/A
Laguna Hold-Fix Optimization	N	N/A	Y[1]	N
Forced Net Replication	Y[2]	N	N	N/A
SLR-Crossing Optimization	Y[1]	Y[1]	Y[1]	Y[1]
Equivalent Driver Rewiring	Y[2]	N	N	N/A

Notes:

1. For UltraScale and UltraScale+ devices only.
2. For Versal devices only.

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 18: **Optimization Options and Values**

phys_opt_design Option	PHYS_OPT_MODIFIED Value
`-fanout_opt`	FANOUT_OPT
`-placement_opt`	PLACEMENT_OPT
`-routing_opt`	MISC_OPT
`-slr_crossing_opt`	SLR_CROSSING_OPT
`-insert_negative_edge_ffs`	INSERT_NEGEDGE
`-restruct_opt`	RESTRUCT_OPT
`-interconnect_retime`	INTERCONNECT_RETIME_OPT
`-lut_opt`	LUT_OPT
`-casc_opt`	CASCADE_LUT_OPT
`-cell_group_opt`	CELL_GROUP_OPT
`-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
`-hold_fix`	HOLD_FIX
`-aggressive_hold_fix`	HOLD_FIX
`-retime`	MISC_OPT
`-force_replication_on_nets`	FORCE_REPLICATION_ON_NETS
`-critical_pin_opt`	MISC_OPT
`-clock_opt`	CLOCK_OPT
`-sll_reg_hold_fix`	MISC_OPT
`-equ_drivers_opt`	EQU_REWIRE_OPT

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 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.

Restructure Optimization

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.

Clock Optimization

By default, clock optimization in Physical Optimization is path-based. This approach focuses on optimizing specific paths within a design. It is useful for improving the timing of critical paths and allows incremental timing updates after each optimization to assess the impact on design performance.

Phase-based clock optimization can be enabled through the `-clock_opt` switch. Phase-based clock optimization divides the optimization process into distinct phases, each targeting different aspects of the design, such as specific block types like URAM, Block RAM, and DSP to take advantage of useful clock skew. It enhances the optimization of paths to these block types by leveraging clock skew, particularly using CLK_MODs in IRI_QUADs to improve negative setup slack. In some designs, there might not be enough output slack available to effectively borrow during this optimization process.

Critical Cell Group Optimization

Optimizes paths of LUT with critical fanin cone group (for Versal only).

Equivalent Driver Rewire Optimization

Optimizes paths by rewiring loads to equivalent drivers. In Versal devices, this optimization is timing driven and nets with fanout less than 100k are considered.

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 fabric or from logic to the UltraRAM cell if it improves the delay on the critical path. Note that this optimization is only done on the output side of the UltraRAM.

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.

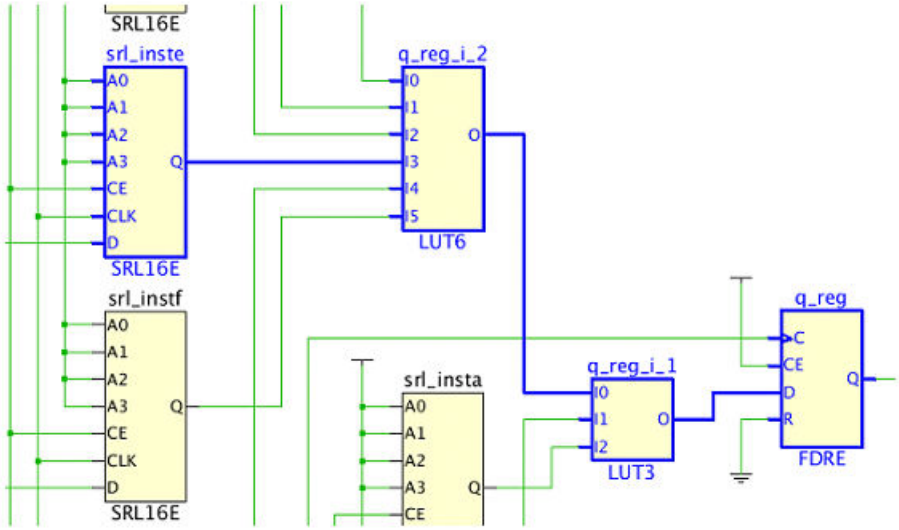
Certain circuit topologies are not optimized:

- 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.

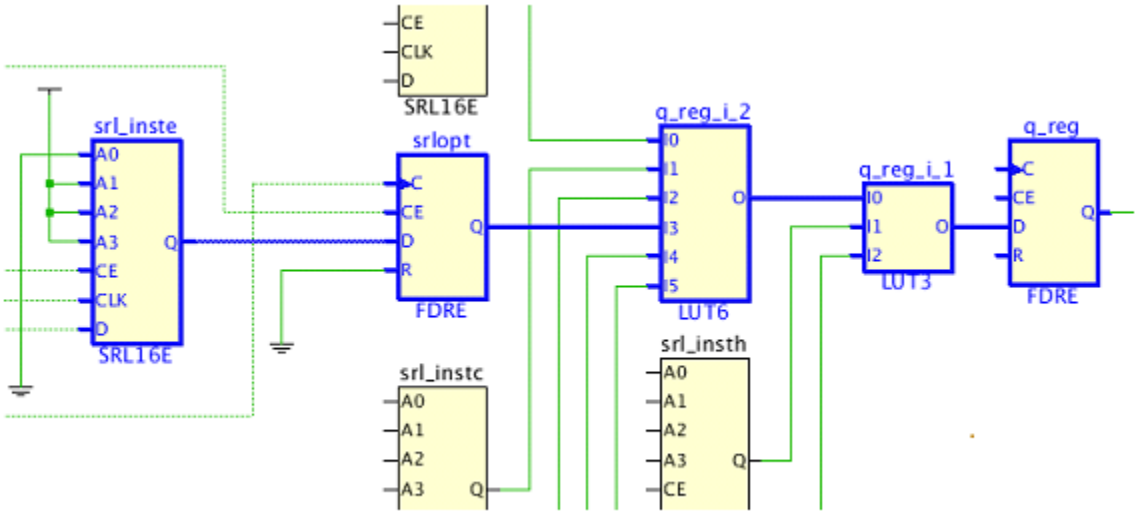
A critical path begins at a shift register (SRL16E) srl_inste, as shown in the following figure.

Figure 26: **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 the following figure.

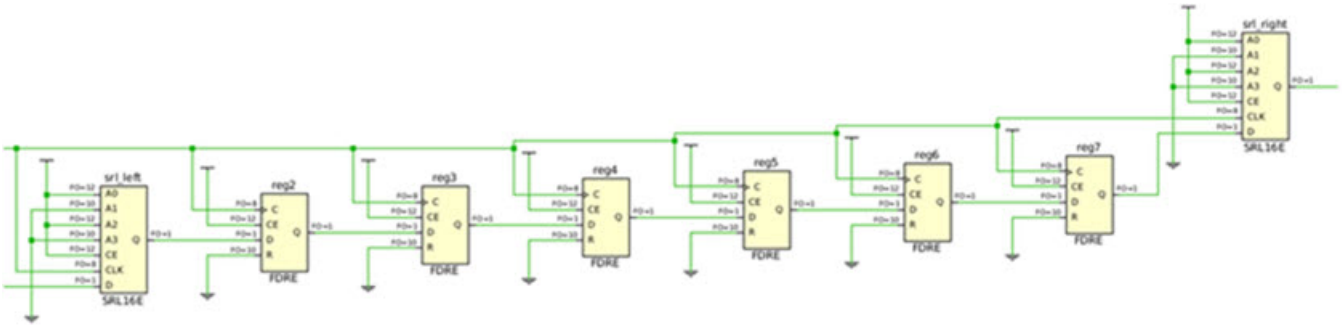
Figure 27: **Critical Path after Shift Register Optimization**



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.

Consider the following logical path, SRL + FFs + SRL, where registers between SRLs have AUTOPIPELINE attributes set.

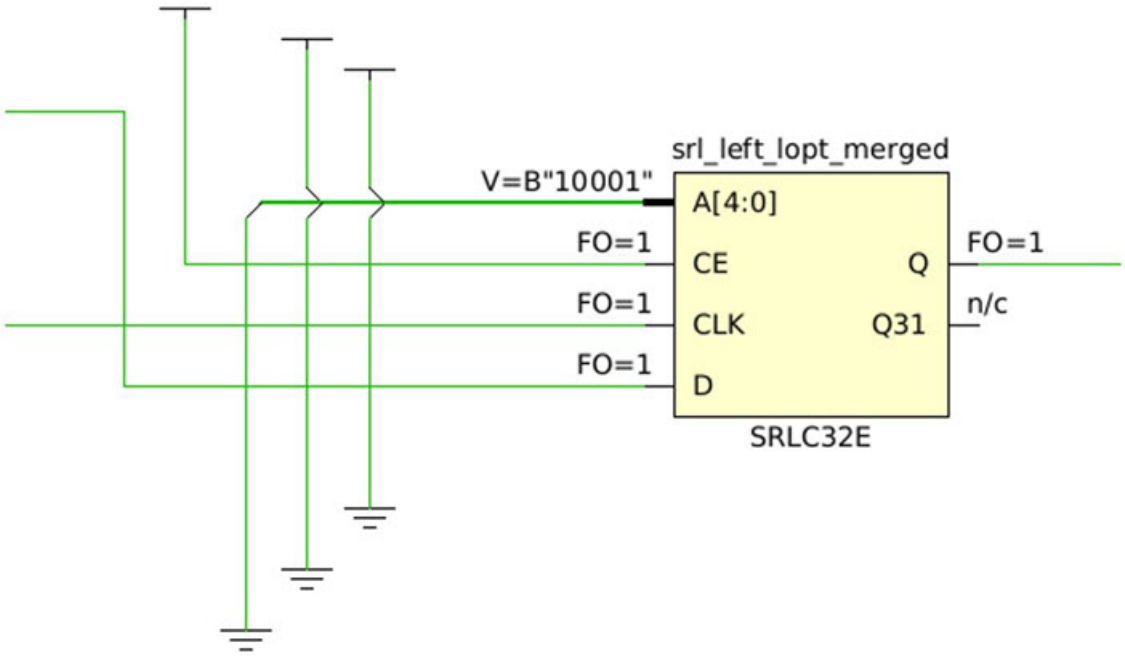
Figure 28: **Auto-pipeline path before Shift Register Optimization**



Although FFs have the AUTOPIPLINE attribute, they are combined into SRL/s after shift register optimization.

As a result, the above circuit is converted into the following SRL cell.

Figure 29: **Auto-pipeline path after Shift Register Optimization**



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 the slack of high-hold violators by increasing the 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.

Interconnect Retiming

Performs interconnect retiming to improve critical path timing by movement or replication of a FF or LUT-FF pair. This is applicable to Versal devices only.

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.

Single LUT Optimization

Performs LUT movement/replication to improve critical path timing.

LUT Cascade Optimization

Performs LUT cascade optimization for creating new LUT cascades or moving existing LUT cascades to improve critical path timing. Applicable for Versal architectures only.

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 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.

Laguna Hold-Fix Optimization

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

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

Path Group Optimization

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

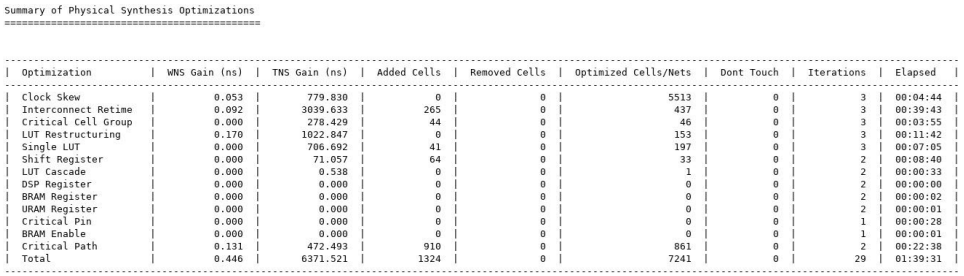
TIP: *Use the `group_path` Tcl command to set up the path groups that are targeted for optimization.*

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.

Figure 30: **Summary of Physical Synthesis Optimizations**



Summary of Physical Synthesis Optimizations

Optimization	WNS Gain (ns)	TNS Gain (ns)	Added Cells	Removed Cells	Optimized Cells/Nets	Dont Touch	Iterations	Elapsed
Clock Skew	0.053	779.830	0	0	5513	0	3	00:04:44
Interconnect Retime	0.092	3039.633	265	0	437	0	3	00:39:43
Critical Cell Group	0.000	278.429	44	0	46	0	3	00:03:55
LUT Restructuring	0.170	1022.847	0	0	153	0	3	00:11:42
Single LUT	0.000	706.692	41	0	197	0	3	00:07:05
Shift Register	0.000	71.057	64	0	33	0	2	00:08:40
LUT Cascade	0.000	0.538	0	0	1	0	2	00:00:33
DSP Register	0.000	0.000	0	0	0	0	2	00:00:00
BRAM Register	0.000	0.000	0	0	0	0	2	00:00:02
URAM Register	0.000	0.000	0	0	0	0	2	00:00:01
Critical Pin	0.000	0.000	0	0	0	0	1	00:00:28
BRAM Enable	0.000	0.000	0	0	0	0	1	00:00:01
Critical Path	0.131	472.493	910	0	861	0	2	00:22:38
Total	0.446	6371.521	1324	0	7241	0	29	01:39:31

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] [-insert_negative_edge_ffs]
                 [-restruct_opt] [-interconnect_retime] [-lut_opt] [-
casc_opt]
                 [-cell_group_opt] [-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 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.

The phys_opt_design -clock_opt command additionally focuses on taking advantage of useful clock skew for COE blocks, targeting BRAM, URAM, and DSP blocks. The process is executed in the pre-route phys_opt_design stage as an extension of the `clock_opt` switch, aimed at optimizing COE paths.

- **Clock Skew Application:** phys_opt_design takes advantage of useful skew using CLK_MODs in IRI_QUADs driving the macro clock pins to improve negative setup slack

Phase Based Optimization

- During phase-based clock optimization iterations, specific block types like URAM, BRAM, and DSP are optimized to take advantage of useful clock skew. PCIe blocks are not currently optimized during the phase-based execution but are optimized during path-based execution of phy_opt_design.
- Phase-based optimization involves dividing the optimization process into distinct phases, each targeting different aspects or areas of the design.

Usage

phys_opt_design -clock_opt

Path-Based Clock Optimization

Timing path based optimizations take a failing timing path and try multiple optimizations to improve the performance of the path. This approach is particularly useful for improving the timing of critical paths that might be limiting the overall performance of the design. It allows for incremental updates to timing after each optimization, which helps in assessing the impact of changes on the design's performance.

Usage

phys_opt_design

In some designs, there might not be enough output slack available to effectively borrow during this optimization process. This can limit the extent of improvements achievable for PCIe and other COE blocks.

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. Consider using this directive if you are requiring to run several iterations of phys_opt_design to achieve improved WNS.
- **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.
- **RQS:** Instructs phys_opt_design to select the phys_opt_design directive specified by the report_qor_suggestion strategy suggestion. Requires an RQS file with a strategy suggestion to be read in prior to calling this directive.

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 section Synthesis Attributes in the *Vivado Design Suite User Guide: Synthesis* (UG901).

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 recommended approach for managing MARK_DEBUG usage is the `config_flows -mark_debug` option, which allows you to control optimization of objects with MARK_DEBUG without modifying source files or constraints. The following three values are supported:

- **enable:** Do not optimize MARK_DEBUG nets. This is a default value.
- **disable:** Allow both synthesis and implementation to freely optimize MARK_DEBUG nets.
- **synthesis_only:** Synthesis will not optimize MARK_DEBUG nets so that they are available at the beginning of implementation, but MARK_DEBUG nets can be freely optimized during implementation.

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 *not* cumulative. Each `phys_opt` run has a different phys_opt report that only accounts for the changes made during that particular run 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.

Figure 31: **Fanout Optimization Report**

```
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   |
```

Interactive Physical Optimization

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.

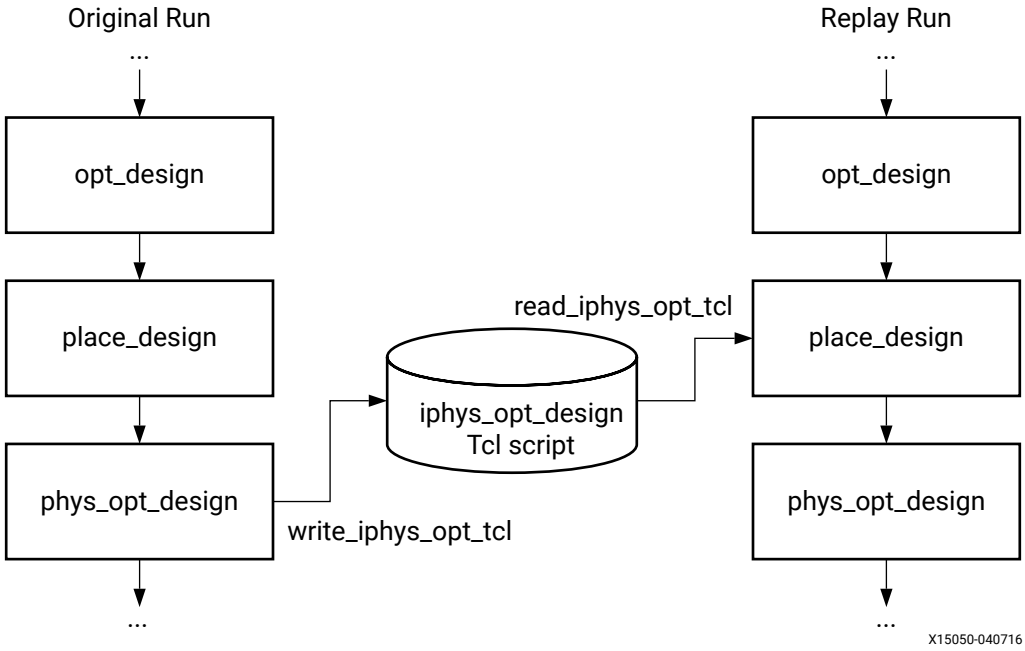
Interactive physical optimization can be used in two ways:

- 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 32: **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.

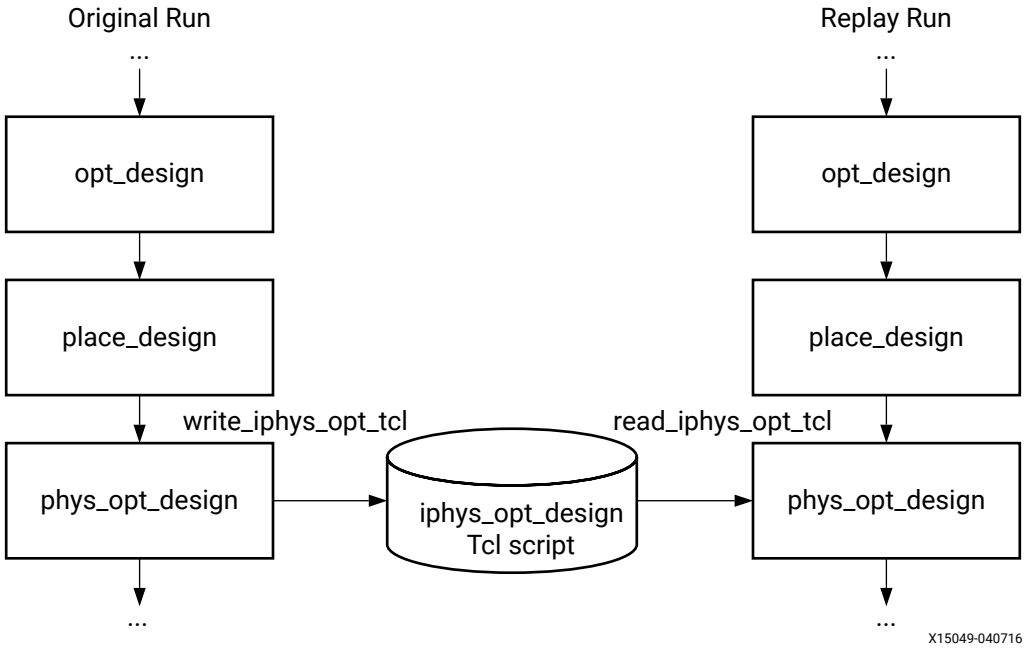
The same design and constraints are used for the replay run. Before `place_design` runs, the `read_iphys_opt_tcl` command processes the `iphys_opt_design` command script and applies the netlist changes from the original run. As a result of the netlist changes, the design in the replay run might be more 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 33: **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] [-replicate_cell]
                   [-placement_opt] [ -restruct_opt] [-forward_retime]
                   [-backward_retime] [-dsp_register_opt]
                   [-bram_register_opt]
                   [-uram_register_opt] [-shift_register_opt]
                   [-shift_register_to_pipeline] [-auto_pipeline]
                   [-pipeline_to_shift_register] [-critical_pin_opt]
                   [-restruct_opt] [-equ_drivers_opt]
                   [-include_skipped_optimizations] [-create_bufg]
                   [-insert_negative_edge_ffs] [-hold_fix]
                   [-slr_crossing_opt] [-quiet]
                   [-verbose] [<input>]
```

The `read_iphys_opt_tcl` 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`
- `-restruct_opt`
- `-dsp_register_opt`
- `-bram_register_opt`
- `-uram_register_opt`
- `-shift_register_opt`

- `-insert_negative_edge_ffs`
- `-slr_crossing_opt`
- `-critical_pin_opt`
- `-replicate_cell`
- `-forward_retime`
- `-backward_retime`
- `-shift_register_to_pipeline`
- `-auto_pipeline`
- `-pipeline_to_shift_register`
- `-equ_drivers_opt`
- `-create_bufg`

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 compile time, use the `read_iphys_opt_tcl` command instead.*

Syntax

```
iphys_opt_design  [-fanout_opt] [-critical_cell_opt] [-replicate_cell]
                  [-reconnect] [-placement_opt] [-forward_retime]
                  [-backward_retime] [-net <arg>] -cluster <args>
                  -place_cell <args> [-dsp_register_opt]
                  [-bram_register_opt]
                  [-uram_register_opt] [-shift_register_opt] [-cell <arg>]
                  [-packing] [-unpacking] [-port <arg>] [-critical_pin_opt]
                  [-restruct_opt] [-equ_drivers_opt] [-skipped_optimization]
                  [-create_bufg] [-insert_negative_edge_ffs] [-hold_fix]
                  [-slr_crossing_opt] [-shift_register_to_pipeline]
                  [-auto_pipeline] [-pipeline_to_shift_register] [-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 compile time.
- 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 compile time penalty 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.

Critical warnings in the initial routing phase that are unlikely to converge later on in the flow are treated as conditions for early exit. If a design exhibits one such condition, the router exits after the initial routing phase to reduce compile time. The route state of the design after early exit does not offer any more information than the pre-route checkpoint. A design in this state is not suitable for analysis as routing deposit information is not available. Pre-route designs can be used for further debugging. Following are some conditions for early exit:

- Placement or design constraint issues that might lead to routing failures
- Blockages between source and destination, possibly due to fixed routing constraints
- If number of nets crossing SLR boundary exceeds maximum capacity of a device
- Global congestion level of eight or more

Upon encountering one of the above conditions, you will see the following message:

- **ERROR:** [Route 35-4445] route_design is terminated due to errors or critical warnings issued before and during initial routing. The issues reported cannot be resolved later in `route_design`.
- **Resolution:** Review previous errors or critical warnings messages to determine the reason for router exit during initial routing stage. To know the state of route database, run `report_route_status`.
- **INFO:** [Route 35-17] Router encountered errors. Check the log file for details.

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 router 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 section Checking That Your Design is Properly Constrained in UltraFast Design Methodology Guide for FPGAs and SoCs (UG949).*

Router Timing Summary

At the end of the routing process, the router reports an estimated timing summary calculated using routing delays. However, to improve runtime, the router uses incremental timing updates rather than doing the full timing computation to calculate the timing summary. Consequently, the estimated WNS can be pessimistic by a few picoseconds compared to signoff 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 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] [-eco]
             [-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 compile time 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 compile time for better performance.
- **`RuntimeOptimized`:** Run fewest iterations, trade higher design performance for faster run time.
- **`AlternateCLBRouting`:** Chooses alternate routing algorithms that require extra compile time but might 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.
- **`RQS`:** Instructs route_design to select the directive specified by the report_qor_suggestion strategy suggestion. Requires an RQS file with a strategy suggestion to be read in prior to calling this directive. This option is not available for Versal.

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.

- **`-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.
- **`-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.
- **`-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.
- **`-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.
- **`-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`.
- **`-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.

- **`-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.

- **`-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.

- **`-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`.

- **`-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.

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

- **`-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.

- **`-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.

- `-no_timing_driven`: This option disables timing-driven routing and is used primarily for testing the routing feasibility of a design.
- `-eco`: This option is used with incremental mode to get a shorter compile time 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.

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 compile time 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 19 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. In addition, the router also generates a `tight_setup_hold_pins.txt` text file that contains a list of the endpoint pins and the launch and capture clock.

TIP: *Use the **Pin** value from `tight_setup_hold_pins.txt` file and use the following for improving the timing paths.*

```
report_timing -delay_type min_max -nworst 4 -to [get_pins <pin>]
-name timing_set_hold_pin_1
```

- 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|   32x32|      0.89|   32x32|      1.00|   32x32|      0.66|
|_____________|________|__________|________|__________|________|__________|
|        SOUTH|   16x16|      0.68|   32x32|      0.75|   16x16|      0.53|
|_____________|________|__________|________|__________|________|__________|
|         EAST|     4x4|      0.04|     8x8|      0.09|     4x4|      0.10|
|_____________|________|__________|________|__________|________|__________|
|         WEST|     8x8|      0.18|     8x8|      0.09|   16x16|      0.50|
|_____________|________|__________|________|__________|________|__________|
```

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).

***Note*:** In some cases, when the design is congested, router completes while leaving a few nets that are not fully routed. In such cases, Vivado issues a critical warning instead of an error. This is done to ensure that the flow does not stop abruptly and a dcp is available for further debugging.

Users can change the severity of the error message using the following command: `set_msg_config -id "Route 35-1" -new_severity "ERROR"`

In order for the message severity change to take effect, it should be applied before running `route_design` for any message that is printed out during `route_design`. In a project flow, this can be added to the pre-route Tcl script.

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 section Routing in the *UltraFast Design Methodology Guide for FPGAs and SoCs* (UG949).

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 19: **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).

Incremental Implementation

***Note*:** This flow is not supported for Versal.

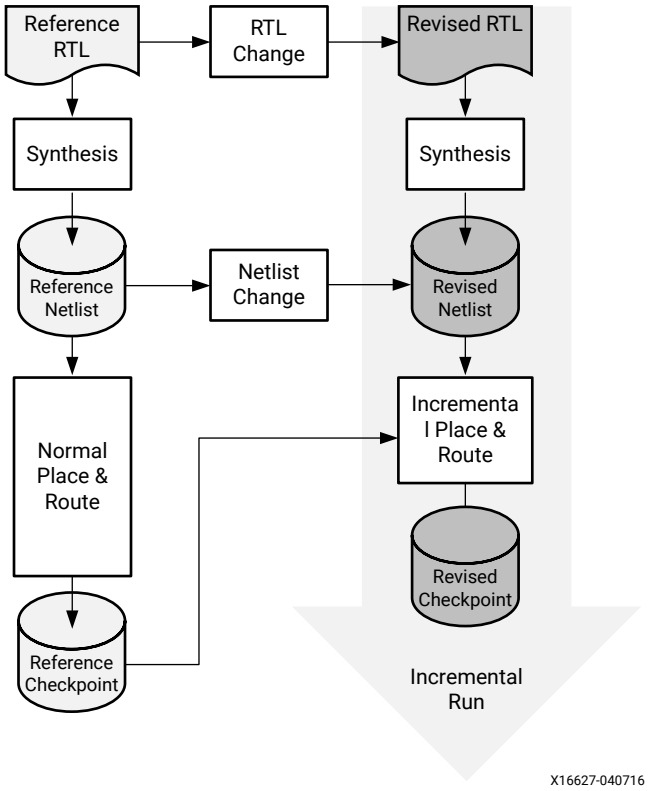
Incremental Implementation refers to the implementation phase of the incremental compile design flow that:

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

A diagram of the incremental implementation design flow is provided in the following figure.

This diagram also illustrates 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).

Figure 34: **Incremental Compile Design Flow**



Incremental Implementation Terminology

The incremental flow uses a variant from a previous run as well as the current design variant. In order to maintain clear understanding the following terminology is used:

Reference Design

The reference design is preferably a fully routed checkpoint from a recent iteration of the same design. To use a different variant of a design, it is important that the hierarchy names from the reference design match the incremental design. Whilst it is possible to use a placed checkpoint as a reference, there will be reduced benefits when compared to a routed checkpoint; timing will not be as consistent and compile time will be higher.

The reference design must match the device and it is recommended to match the tool version but not a strict 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 but these changes should be typically < 5%. Prior to issuing the `read_checkpoint -incremental` command, there is no knowledge that the incremental implementation flow is being used. Therefore it is important to not introduce significant netlist changes by changing `synth_design` or `opt_design` tool options when compared with the reference design.

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.

Running Incremental Place and Route

With the incremental design loaded in memory, the incremental flow is initiated by loading the reference design checkpoint using the `read_checkpoint -incremental <dcp>` command. 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. It is possible to have partially-reused routes.

Placement and routing information that is reused initially can be discarded throughout the flow if it improves the performance or aids routability.

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.

Incremental Mode

When incremental mode is selected by the user, the tool might still not run the full incremental flow if it determines the design change to too much. Incremental implementation is typically run if cell reuse is above 80%.

The flow selected will be either:

- A full place and route using incremental optimized algorithms. Placement and routing are reused as much as is possible. Target WNS is determined by a combination of both the reference checkpoint and the directive. Directives are selected based on the directive supplied to the `read_checkpoint -incremental -directive <directive>` switch.
- A full place and route using the default algorithms. Placement and routing are not reused. Target WNS is always 0.000. Directives are taken from the directive switch supplied to either the `place_design -directive <directive>` or `route_design -directive <directive>` commands.

This decision is taken after the design modifications and cell matching process during the `read_checkpoint -incremental` command. As this assessment is made after changes to the design are made to improve matching so it is not the same as running purely a default flow as the changes are persistent.

When incremental mode is entered the following message can be seen:

```
INFO: [Place 46-42] Incremental Compile tool flow is being used. Default
place and route algorithms are bypassed.
```

Automatic Incremental

Automatic Incremental Implementation is designed to leverage the faster compile times of incremental implementation whilst not impacting quality of results such as WNS. It is a subset of the full incremental flow with tighter controls to ensure performance does not degrade. It works to the following criteria:

1. Updating the reference checkpoint only when WNS is >=-0.250 ns. This is only actively managed in project mode. In non project flow, users must follow the script provided below.
2. Setting higher targets for WNS and reuse during the `read_checkpoint -incremental` phase.
   - 94% cell matching
   - 90% net matching
   - WNS >= -0.250 ns

The flow is activated using the following command:

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

When updating the checkpoint, the following script will ensure that WNS has not degraded beyond acceptable limits:

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

Incremental Directives

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

```
read_checkpoint -incremental -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.*

Further Options

The following options are available when using the `read_checkpoint -incremental` command.

-auto_incremental Option

This enables the automatic incremental flow described in Automatic Incremental.

-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.

Examples

The following are examples of the `-fix_objects` usage:

- To reuse all objects and fix only Block Memory placement:

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

- To reuse all objects and fix only DSP placement:

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

-force_incr Option

The `-force_incr` option can be used to force the incremental implementation flow irrespective of the incremental criteria checks. When not specified the incremental implementation flow might exit and continue in non-incremental or default flow.

This option can be used instead of modifying the incremental implementation configurations values to update the minimum thresholds for cell matching, net matching, and WNS in the automatic incremental flow.

***Note*:** `-force_incr` is only used along with the -incremental switch.

Incremental Compile Analysis

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

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 seven sections.

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     |               Yes |
| Incremental Directive | RuntimeOptimized |
| Target WNS           |               0.0 |
| QoR Suggestions      |                 0 |
+----------------------+------------------+
```

Reuse Summary

This contains an overview of the cells, nets, pins, and ports that are reused. An example is:

```
2. Reuse Summary
+-------+----------+----------------+---------------+---------+-------+
|  Type | Matched% | Initial Reuse% | Current Reuse% | Fixed%  | Total |
+-------+----------+----------------+---------------+---------+-------+
| Cells |   100.00 |         100.00 |        100.00 |    0.73 |  5339 |
| Nets  |   100.00 |         100.00 |        100.00 |    0.00 |  3773 |
| Pins  |        - |         100.00 |        100.00 |       - | 23109 |
| Ports |   100.00 |         100.00 |        100.00 |  100.00 |    24 |
+-------+----------+----------------+---------------+---------+-------+
```

The columns provide the following information:

- Matched - Cells that have the same instance name, REF_NAME property or have been deemed a match if some information differs slightly. This is calculated at the end of `read_checkpoint -incremental`
- Initial reuse - Once items are matched, this indicates if the matching information on location was reused or not. Sometimes items are matched not possible to be reused due to illegal connectivity or other design requirements.
- Current reuse - This indicates the reuse at the current design phase. Useful when compared with initial reuse to see how much reuse is lost as you go through the tool flow to generate a legal solution.
- Fixed - items that incremental flow can not touch. When high, the tools might struggle to generate a legal solution.

***Note*:** The titles have been trimmed.

Reference Checkpoint Information

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

- 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:

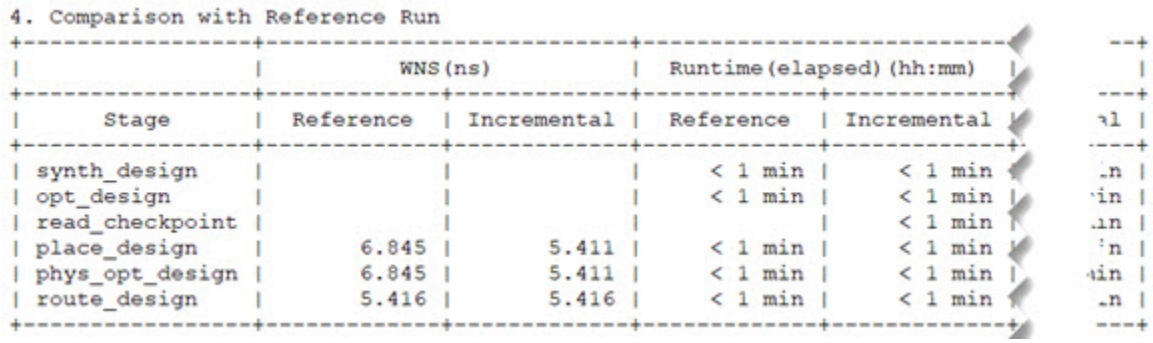
```
3. Reference Checkpoint Information
+----------------+------------------------------+
| DCP Location:  | C:/Temp/mb_preset_routed.dcp |
+----------------+------------------------------+

+--------------------------------+-----------------------------+
|         DCP Information        |            Value            |
+--------------------------------+-----------------------------+
| Vivado Version                 |                      2023.1 |
| DCP State                      |                  POST_ROUTE |
| Recorded WNS                   |                       5.411 |
| Recorded WHS                   |                       0.018 |
| Reference Speed File Version   | PRODUCTION 1.29 05-01-2022 |
| Incremental Speed File Version | PRODUCTION 1.29 05-01-2022 |
+--------------------------------+-----------------------------+
```

Comparison with Reference Run

This contains useful metrics about a comparison with the reference run. From this section you can compare the following:

- Compile time information
- WNS at each stage of the flow
- Tool options at each stage of the flow.

Figure 35: **Reference Run Comparison Example**



4. Comparison with Reference Run

	WNS(ns)		Runtime(elapsed)(hh:mm)		
Stage	Reference	Incremental	Reference	Incremental	
synth_design			< 1 min	< 1 min	
opt_design			< 1 min	< 1 min	
read_checkpoint				< 1 min	
place_design	6.845	5.411	< 1 min	< 1 min	
phys_opt_design	6.845	5.411	< 1 min	< 1 min	
route_design	5.416	5.416	< 1 min	< 1 min	

When using this report it is important to understand that the Incremental run should be compared to the post route numbers of the Reference run as the placement and routing should be reused. For the sections that are not reused, these might contain a mixture of unplaced, placed, or routed information that can impact the accuracy of the metrics reported.

***Note*:** For further understanding, conduct a timing analysis on a checkpoint written out after `read_checkpoint -incremental`.

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. Optimization Comparison With Reference Run
---------------------------------------------

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

+--------------------------+--------+------------+
| iphys_opt_design replay  | Reused | Not Reused |
+--------------------------+--------+------------+
| uram_register_opt        |      1 |          0 |
| replicate_cell           |     60 |          0 |
| critical_cell_opt        |      8 |          0 |
| restruct_opt             |     12 |          0 |
| dsp_register_opt         |     35 |          0 |
| equ_drivers_opt          |     99 |          0 |
| fanout_opt               |     19 |          0 |
| shift_register_opt       |      0 |         14 |
| bram_register_opt        |      1 |          0 |
| hold_fix                 |      1 |          0 |
| reconnect                |     20 |          0 |
+--------------------------+--------+------------+
```

***Note*:** Physical optimizations that have failed to replay result in lower reuse can impact the likelihood of maintaining the current timing picture.

Command Comparison with Reference Run

This section contains the commands executed for flow command comparison. An example is:

```
6. Command Comparison with Reference Run
6.1 Reference:
synth_design-verilog_define default::[not_spec      :ku5p-ffvb676-2-e
opt_design
place_design
phys_opt_design
route_design

6.2 Incremental:
synth_design-verilog_define default::[not_speci      .u5p-ffvb676-2-e
opt_design
read_checkpoint -directive RuntimeOptimized  -i.      preset_routed.dcp
place_design
phys_opt_design
route_design
```

***Note*:** In Incremental flow, `iphys_opt_replay` might not replay all optimizations from the reference run. Any Non-reused optimization has an impact on the cell or net reuse.

Pay particular attention to the commands used prior to read_checkpoint -incremental to confirm they are the same and reuse is maximized.

Non-reuse Information

This contains metrics about what was not reused and why. The following is an example:

```
7. Non Reuse Information
+-----------------------+------+
|          Type         |   %  |
+-----------------------+------+
| Non-Reused Cells      | 0.00 |
| Partially reused nets | 0.00 |
| Non-Reused nets       | 0.00 |
| Non-Reused Ports      | 0.00 |
+-----------------------+------+
```

Cell Based Reporting

The `-cells` option limits the reuse reporting to the list of given cells instead of reporting reuse of the entire design. It can be hierarchical cells or specific leaf cells.

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

When `-cells` is used:

- The totals used to calculate matched %'s are based on the total number of cells returned by the list of cells. When not used the totals are the number of cells in the full design
- Nets are assumed to be the nets attached to the list of cells provided

In the following example, the report is limited to only report reuse on block RAM cells:

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

Hierarchical Reuse Reporting

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.

```
Hierarchical Implementation Reuse Summary
1. Summary
+-----------------------+--------------------+--------+-----+----------------------+------+-----+
|       Instance        |       Module       | Reused | New | Discarded(Illegal)*  | Disc | *** |
+-----------------------+--------------------+--------+-----+----------------------+------+-----+
| mb_preset_wrapper     |              (top) |   5339 |   0 |                    0 |      |   0 |
|   (mb_preset_wrapper) |              (top) |     37 |   0 |                    0 |      |   0 |
|   mb_preset_i         |                 mb |   5302 |   0 |                    0 |      |   0 |
|     axi_gpio_0        |       mb_axi_gpio_0 |    159 |   0 |                    0 |      |   0 |
|     axi_gpio_1        |       mb_axi_gpio_1 |    128 |   0 |                    0 |      |   0 |
|     axi_iic_0         |        mb_axi_iic_0 |    849 |   0 |                    0 |      |   0 |
|     axi_timer_0       |      mb_axi_timer_0 |    629 |   0 |                    0 |      |   0 |
|     axi_uartlite_0    |  mb_axi_uartlite_0 |    224 |   0 |                    0 |      |   0 |
|     clk_wiz_1         |       mb_clk_wiz_1 |      4 |   0 |                    0 |      |   0 |
|     mdm_1             |           mb_mdm_1 |    223 |   0 |                    0 |      |   0 |
|     mb_0              |     mb_microblaze_0 |   2222 |   0 |                    0 |      |   0 |
|     mb_0_axi_intc     |         mb_axi_intc |    399 |   0 |                    0 |      |   0 |
|     mb_0_axi_periph   |           mb_periph |    345 |   0 |                    0 |      |   0 |
|     mb_0_local_memory |  mb_0_local_memory |     59 |   0 |                    0 |      |   0 |
|     mb_0_xlconcat     |         mb_xlconcat |      0 |   0 |                    0 |      |   0 |
|     rst_clk_wiz_1_100M | mb_rst_clk_wiz_100M |     61 |   0 |                    0 |      |   0 |
+-----------------------+--------------------+--------+-----+----------------------+------+-----+
```

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 mb_preset_wrapper with a cumulative reuse total of 5339 cells with 0 new cells. The row with mb_preset_wrapper in parentheses shows the cell reuse status contained within mb_preset_wrapper and but not its submodules. Of the 5339 cells, only 37 are within mp_preset_wrapper and the remainder are within its submodules.

There are five 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
```

```
+----------------------+--------+--------+-----+----------------------+------
|       Instance       | Module | Reused | New | Discarded(Illegal)*  | Disca
+----------------------+--------+--------+-----+----------------------+------
|   mb_preset_wrapper  |  (top) |   5339 |   0 |                    0 |
+----------------------+--------+--------+-----+----------------------+------
```

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

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.

Figure 36: **Incremental Reuse Summary in Vivado**

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			

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: *AMD 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 section Incremental Flows in the UltraFast Design Methodology Guide for FPGAs and SoCs (UG949).*

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`.

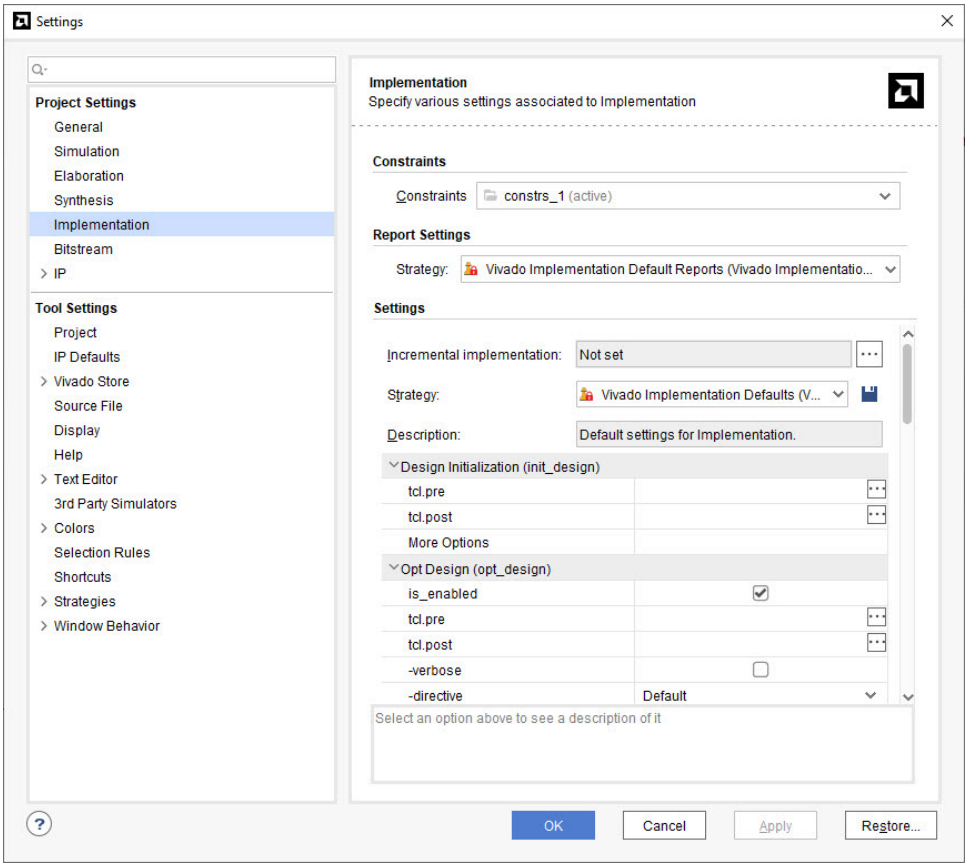
Using Incremental Implementation in Project Mode

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:

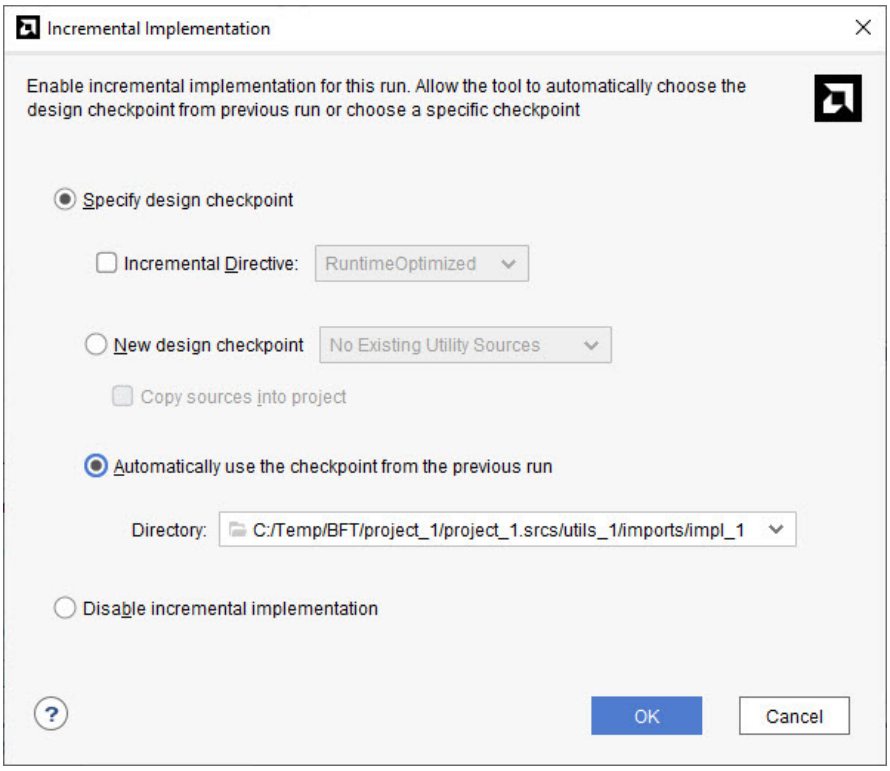
1. Right-click a run in the Design Runs window.
2. Click **Set Incremental Implementation** from the context menu.

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 dialog box.



To enable the incremental flow:

1. Select the **Specify design checkpoint** radio button.
2. Next you have the option to set the directive. When unselected, this is set to the default value **RuntimeOptimized**, other values are **TimingClosure** and **Quick**.
3. Select whether you want a static reference checkpoint using the **New design checkpoint** option or one that automatically updates if a newer suitable reference checkpoint is available by selecting the **Automatically use the checkpoint from the previous run** option. This also allows a directory to be selected so that the checkpoint can be stored in an area outside the project structure to make it more version control friendly.

The following is an example of Tcl commands that can set up the incremental flow to use the TimingClosure directive and reference a static checkpoint:

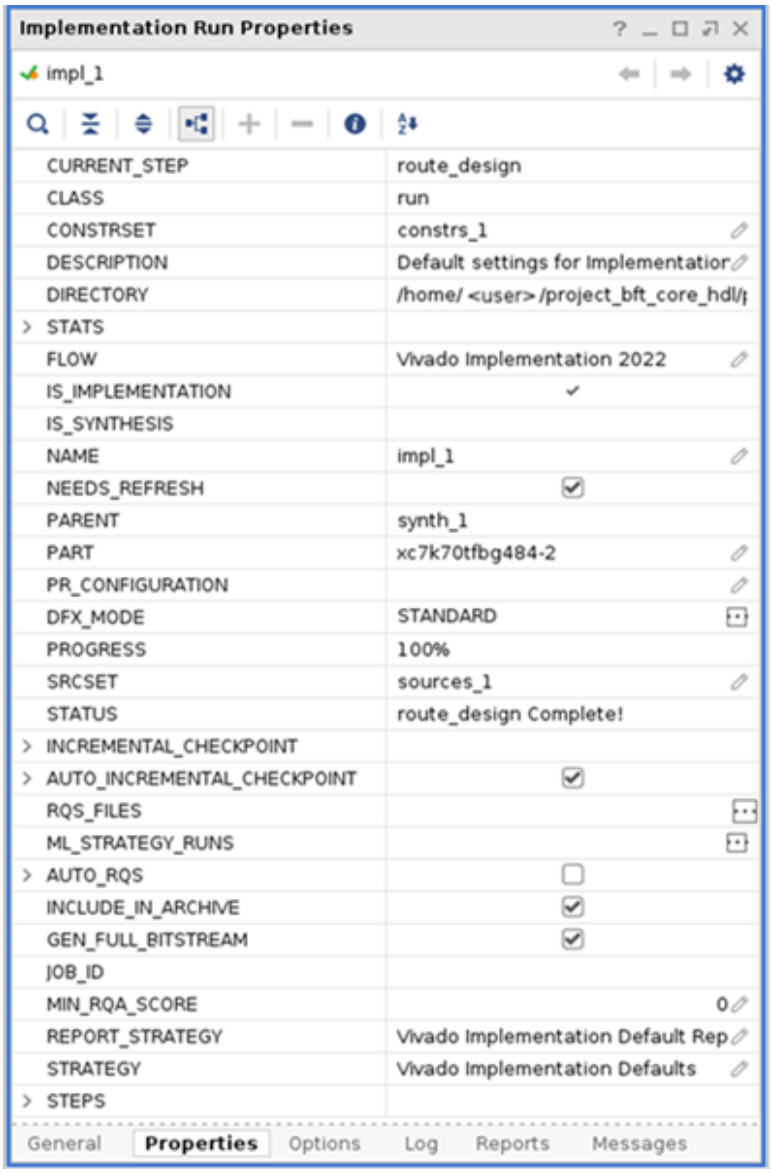
```
set_property AUTO_INCREMENTAL_CHECKPOINT 0 [get_runs impl_1]
set_property incremental_checkpoint.directive TimingClosure [get_runs
impl_1]
add_files -fileset utils_1 -norecurse ./top_routed.dcp
set_property incremental_checkpoint ./top_routed.dcp [get_runs impl_1]
```

The following is an example of the Tcl commands required to set up the incremental flow to use the RuntimeOptimized directive and automatically update the checkpoint:

```
set_property AUTO_INCREMENTAL_CHECKPOINT 1 [get_runs <run_name>]
set_property AUTO_INCREMENTAL_CHECKPOINT.DIRECTORY <directory> [get_runs
impl_1]
set_property incremental_checkpoint.directive RuntimeOptimized [get_runs
impl_1]
```

To apply Incremental Implementation Controls

1. Select the Implementation run
2. In the Implementation Properties under **INCREMENTAL_CHECKPOINT** use the **MORE_OPTIONS** to add the control switches



To disable incremental compile for the current run (or clear the reference to start over again without a reference checkpoint in Automatic mode), do one of the following:

- Select **Disable incremental compile** in the Incremental Implementation dialog box, or
- Run the following command in the Tcl Console:

```
set_property AUTO_INCREMENTAL_CHECKPOINT 0 [get_runs <run_name>]
set_property incremental_checkpoint "" [get_runs impl_1]
```

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
```

Reusing Synthesis Results

The performance of incremental compile can be correlated to the amount of change in synthesis. Synthesis results that are reused limit the amount of change and hence improve the performance of the overall incremental compile flow.

Vivado Synthesis

Vivado synthesis can be run incrementally with little setup required. This requires a checkpoint to be read before synthesis. It is setup automatically when running in project mode but for not project mode it must be added to scripts.

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

IP and Block Designs are automatically synthesized in out of context mode and will reuse cached results when available. For more information see Out-of-Context Design Flow in *Vivado Design Suite User Guide: Design Flows Overview* (UG892).

Synplify Synthesis

Synplify can preserve results using the Compile Point flow. 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 push button 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.

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
close_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.

Factors Impacting Timing Performance

There are many factors that impact the ability of the tool to maintain or improve performance with respect to the reference run. The following list covers the main reasons why a tool might not achieve expected performance

- Low levels of reuse. When reuse is low, having extensive pre-placement of cells can limit the ability to find optimal locations for new logic. It is recommended to use the default flow in these cases.
- Change in the critical path. If the path is made worse, for example by adding more cells, timing is not able to maintain the same performance. For these cases it is recommended to review the change and further optimize.
- Change is in the congested area of the die. When there is limited scope in the design to accept new cells sometimes it can be preferable to find a new solution using the default flow. In these case, users may benefit from running both the default flow and the incremental flow and seeing which performs the best for each case.
- Directive setting. When RuntimeOptimized is set as the directive, the tools will not try to improve beyond what the reference is set to.

Factors Affecting Compile Time Improvement

Factors that can affect compile 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 compile time might be required, and the design might not meet timing.
- The initialization portion of the place and route runtime. In short place and route runs, the initialization overhead of the placer and router might eliminate any gain from the incremental place and route process. For designs with longer compile times, initialization becomes a small percentage of the runtime.

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.
```

Configuring the Incremental Flow

You can configure the incremental flow using the `config_implementation` command. To view the default and current values of this command use the `report_config_implementation` command. To update these values use the `config_implementation` command. Following is an example:

```
report_config_implementation
config_implementation { {incr.ignore_user_clock_uncertainty true} }
```

***Note*:** You can update more than one element at a time by grouping key value pairs in the same method shown above within the outer brackets.

You can configure the following:

- Minimum thresholds for cell matching, net matching, and WNS in the automatic incremental flow.
- Behavior of both synthesis and implementation when the automatic incremental flow criteria is not met. This check happens at the beginning of the synthesis run and during `read_checkpoint -incremental` for implementation. It can be set to `Terminate` which stops the flow or `SwitchToDefaultFlow` which exits the incremental flow but continues with default flow settings.
- Whether the flow ignores user clock uncertainty constraints that are typically used to overconstrain the placer and force closer placement.

Chapter 3

Analyzing and Viewing Implementation Results

Monitoring the Implementation Run

Monitoring the implementation run allows you to:

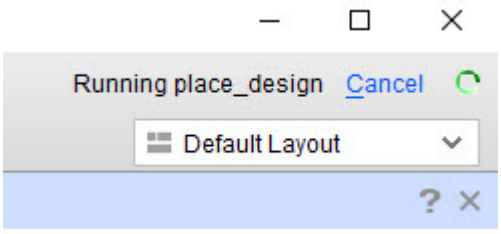
- Read the compilation information.
- Review warnings and errors in the Messages window.
- View the Project Summary.
- Open the Design Runs window.

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

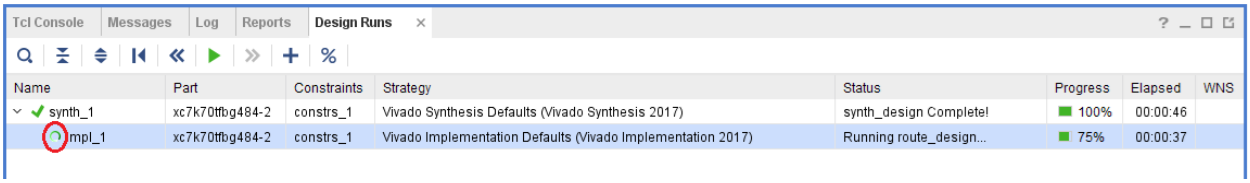
Viewing the Run Status Display

You can display the status of a run that is in progress 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 AMD Vivado™ IDE, as shown in the following figure. The run status indicator displays a scrolling bar to indicate that the run is in process. You can click **Cancel** to end the run.



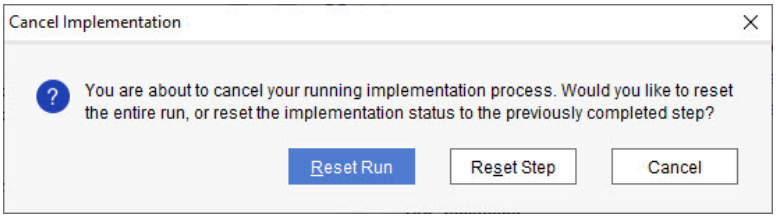
- You can also find a run status indicator in the Design Runs window, as shown at the bottom left of the following figure. It displays a circular arrow (noted in red in the figure) to indicate that the run is in process. You can select the run and use the Reset Run command from the popup menu to cancel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



Canceling or Resetting the Run

If you cancel an in-progress run using **Cancel** or **Reset Run**, Vivado IDE prompts you to delete run files created during the canceled run

Figure 37: **Cancel Implementation**



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

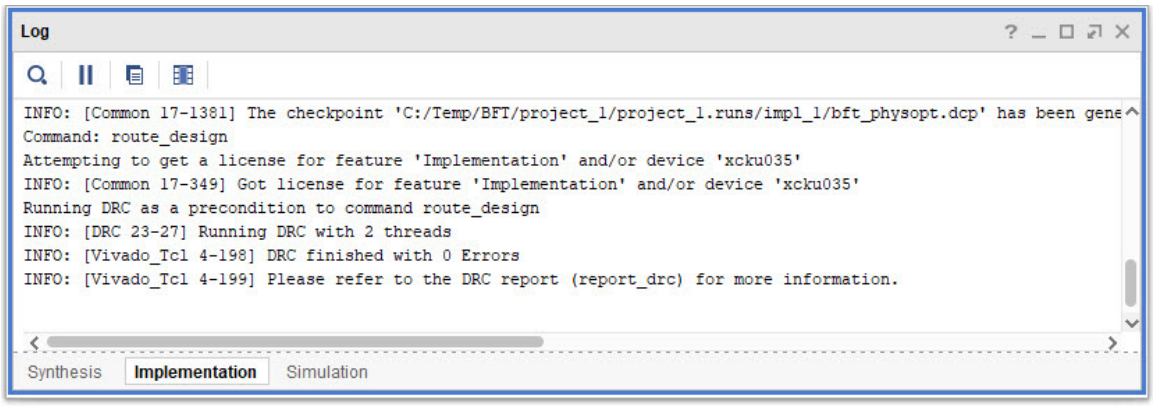
RECOMMENDED: *Delete any data created as a result of a canceled run to avoid conflicts with future runs.*

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 in the following figure, can help you understand where different messages originate to aid in debugging the implementation run.

Figure 38: **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.

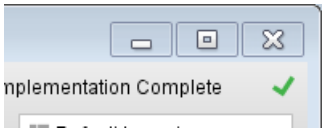
The project status displays in the Project summary and the Status bar. It allows you to immediately see the status of a project when you open the project, or while running the design flow commands, including:

- RTL elaboration
- Synthesis
- Implementation
- Bitstream generation

Viewing Project Status in the Project Status Bar

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

Figure 39: **Project Status Example**



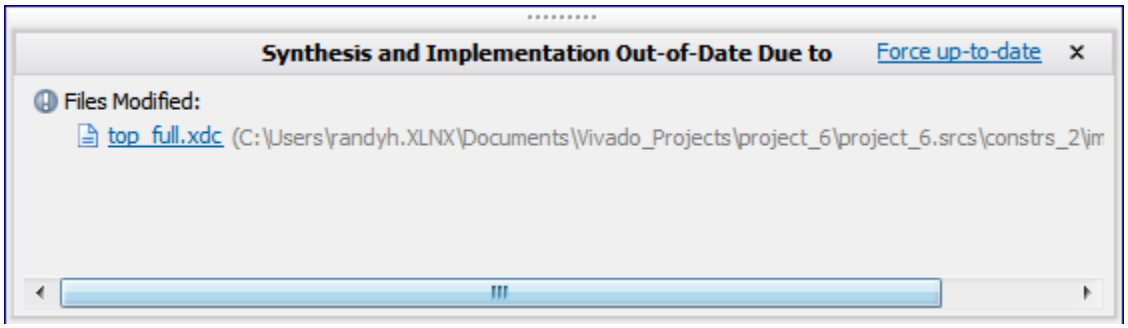
As the run progresses through the Synthesize, Implement, and Write Bitstream commands, the Project Status Bar changes to show either a successful or failed attempt. Failures display in red text.

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.

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 40: **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: *You can also access Force Up-to-Date from the Design Runs window pop-up menu 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 implementation results.

- Is the design fully placed and routed, or are there issues that need to be resolved?
- Have timing constraints and design requirements been met, or are additional changes required to complete the design?
- Are you ready to generate the bitstream for the AMD 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:
   - In Project Mode, open the implemented design for further analysis.
   - In Non-Project Mode, open a post-implementation design checkpoint.

For more information on analysis of the implemented design, see section Interactive Design Analysis in the IDE in the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906).

Moving Forward in Non-Project Mode

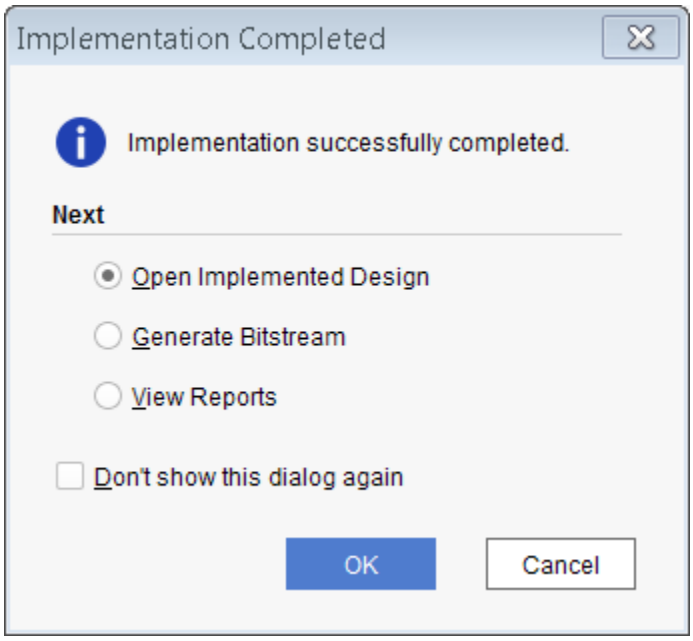
In Non-Project Mode, the Vivado Design Suite generated messages for the design session, and wrote the messages to the Vivado log file (`vivado.log`). Examine this log file and the reports generated from the design data to view an accurate assessment of the current project state.

Moving Forward in Project Mode

In Project Mode, the Vivado Design Suite displays the messages from the log file in the Messages window. It also automates the creation and delivery of numerous reports for you to review.

In Project Mode, after an implementation run is complete in the Vivado IDE, you are prompted for the next step. See the following figure.

Figure 41: **Project Mode - Implementation Completed Dialog Box**



In the Implementation Completed dialog box:

1. Select the appropriate option:
   - **Open Implemented Design:** Imports the netlist, constraints, target part, and place and route results into Vivado IDE for analysis and further work.
   - **Generate Bitstream:** Launches the Generate Bitstream dialog box for 7 series, UltraScale, and UltraScale+ designs, or generates Device Image dialog box for Versal designs. For more information, see *Vivado Design Suite User Guide: Programming and Debugging* (UG908).
   - **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.
2. Click **OK**.

Viewing Messages

IMPORTANT! *Review all messages. The messages might suggest ways to improve your design for performance, power, utilization, 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:

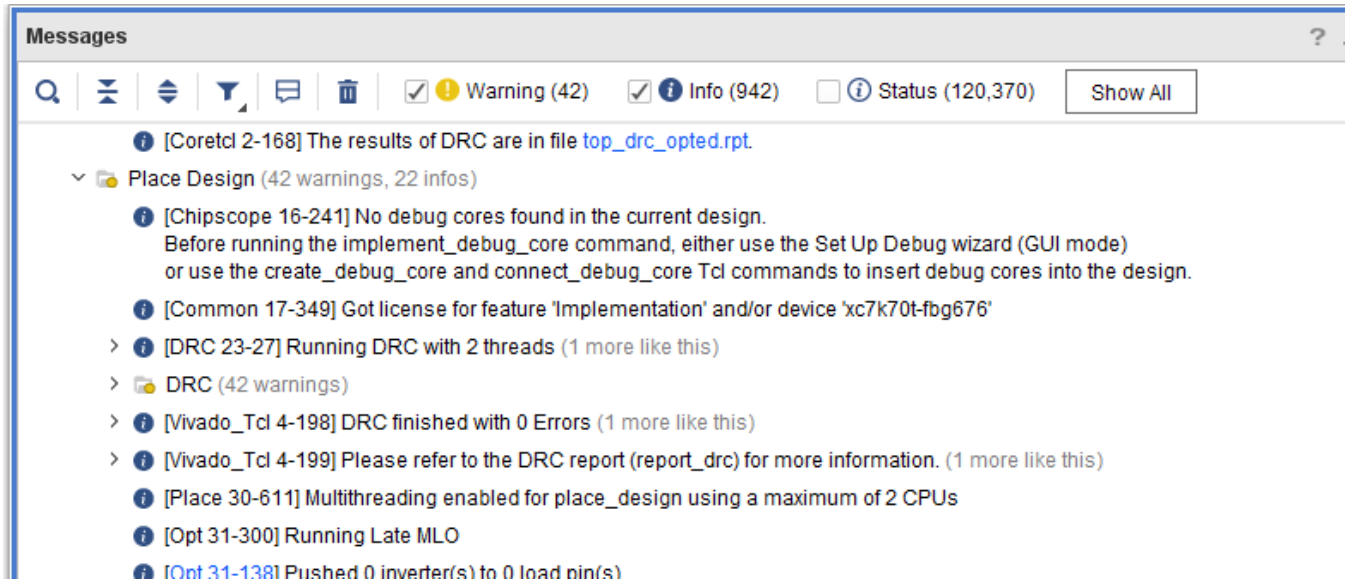
- The commands that you used during a single design session.
- The results and messages from those commands.

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 the following figure, 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 42: **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 pop-up menu to query the AMD Customer Support database for related answer records to the 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 compile 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 did not match 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. Vivado provides the following two options to create the reports

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

Example Tcl Script

```
# 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
```

- Use Tcl command `generate_parallel_reports` to create multiple reports.
- `generate_parallel_reports` command checks interdependencies and creates the reports parallelly.

Example Tcl Script

```
generate_parallel_reports -reports {"report_cdc -details -file
route_report_cdc_0.rpt" "report_control_sets -verbose -file
route_report_control_sets_0.rpt" "report_design_analysis -complexity -file
route_report_design_analysis_0.rpt"}
```

You can generate various reports to analyze a design, but interdependencies prevent some reports from running simultaneously. The tool automatically checks these dependencies and manages the report generation. It allows you to submit a list of reports without being aware of their interdependencies. Below is the interdependency table.

Table 20: **Interdependency Table**

Report_generation name	Execution
report_methodology report_timing_summary report_drc report_timing report_clock_interaction report_cdc report_power report_clock_utilization report_qor_suggestions report_high_fanout_nets report_qor_assesment report_design_analysis report_dfx_summary	Sequential
report_bus_skew report_datasheet	Parallel, with max two threads
report_route_status report_incremental_reuse report_io report_utilization report_control_sets report_ssn report_ram_utilization	Parallel

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 44: Control Sets Report shows an example of a report opened in a Vivado IDE window.

TIP: *Ensure the target report directory exists before running the report; otherwise, you cannot save the file and an error occurs.*

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).

Reporting in Project Mode

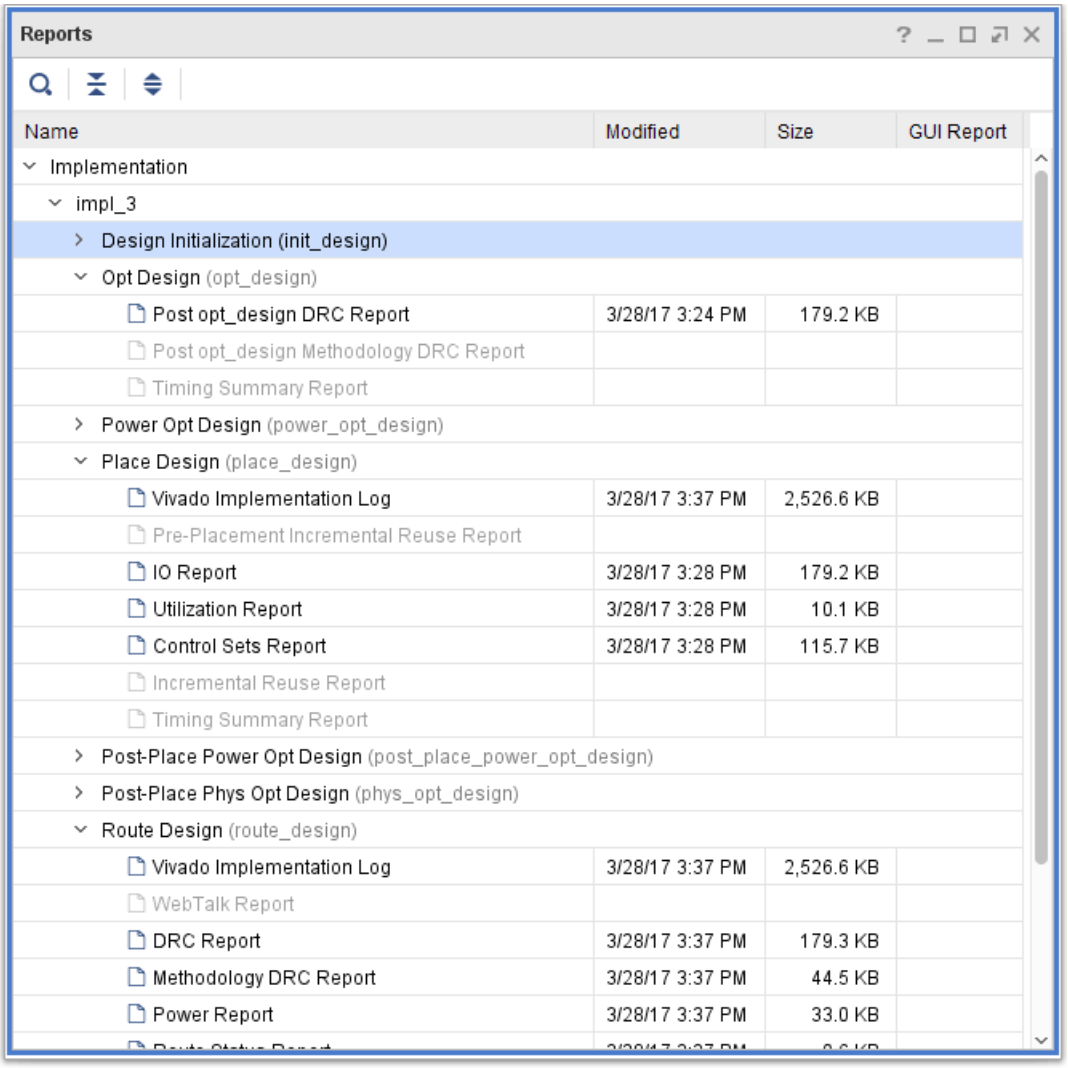
In Project Mode, Vivado automatically generates many reports. View report files in the Reports window, shown in the following figure.

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

1. Select the Reports link in the Project Summary.
2. Select **Windows → Reports**.

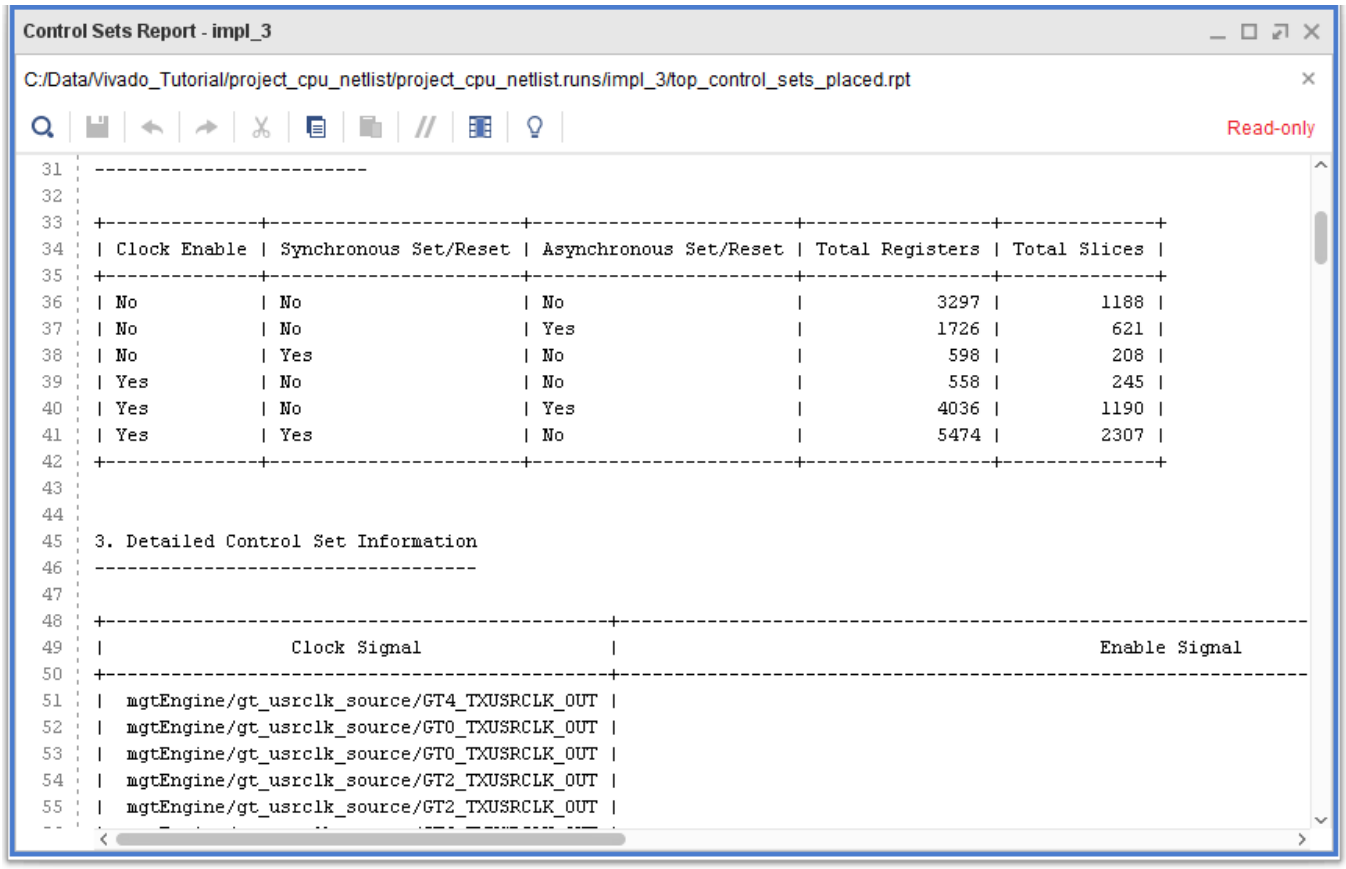
TIP: *The `tcl.pre` and `tcl.post` options of an implementation run let you output custom reports at each step in the process. These reports do not appear in the Reports window, but you can customize them to meet your needs. For more information, see Changing Implementation Run Settings.*

Figure 43: **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 the following figure.

Figure 44: **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 43).

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

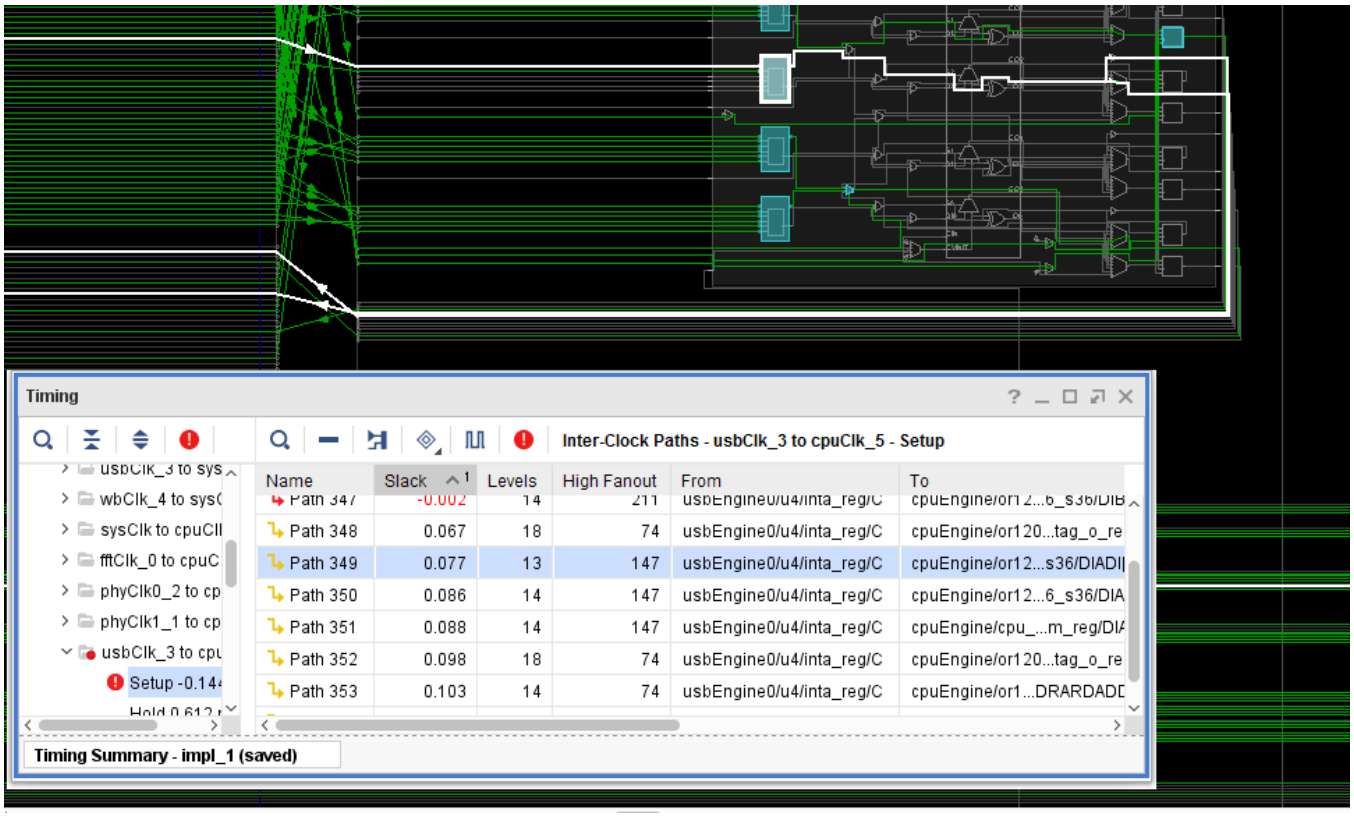
To regenerate the report in the Vivado IDE, select **Tools → Timing → Report Timing Summary**. The resulting report allows you to cross-probe among the various views of the design.

Cross Probing Between Timing Report and Device Window Example

The following figure 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:

- Vivado IDE opens a post-route design checkpoint.
- The Timing Summary report is generated and opened using `report_timing_summary -name`.
- The Routing Resources are enabled in the Device window.
- When you select the timing path in the Timing Summary report, cross probing on the path occurs automatically in the Device window. See the following figure.

Figure 45: **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).

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. They describe 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 imported from an XDC file.

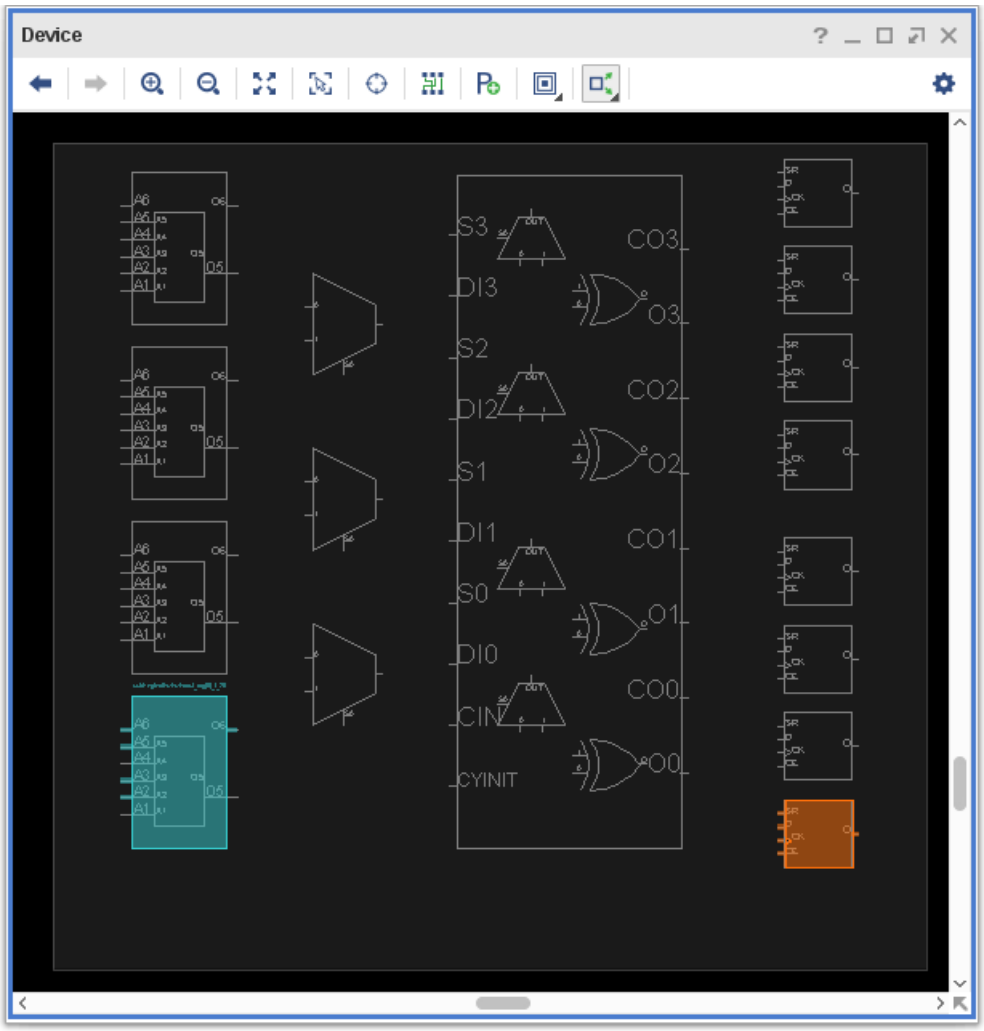
- The Vivado Design Suite treats these placed cells as Fixed.
- Fixed cells are not moved unless directed to do so.
- The FF in the following figure is in orange (default) to indicate that it is Fixed.

Unfixed Cells

Vivado places unfixed cells during implementation, either by 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).
- You can move these cells by the implementation tools as needed in design iterations.
- The LUT in the following figure in blue (default) indicates that it is Unfixed.

Figure 46: **Logic Placed in a Slice**



You can fix both LOCS and BELS. 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 must not go into the XDC.

Fixing Placer-Placed Logic

To fix cells placed by the 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), 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:
  1. Click the **Drag & Drop Modes** toolbar button and select Create BEL Constraint Mode.
  2. 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. 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. Place cells 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: *If you assign logic to an illegal location (for example, due to control-set restrictions for slice flip-flops (FFs)), the Tcl Console reports an error. The tool ignores the assignment.*

Vivado treats cells placed with the `place_cell` Tcl command as Fixed.

Modifying Routing

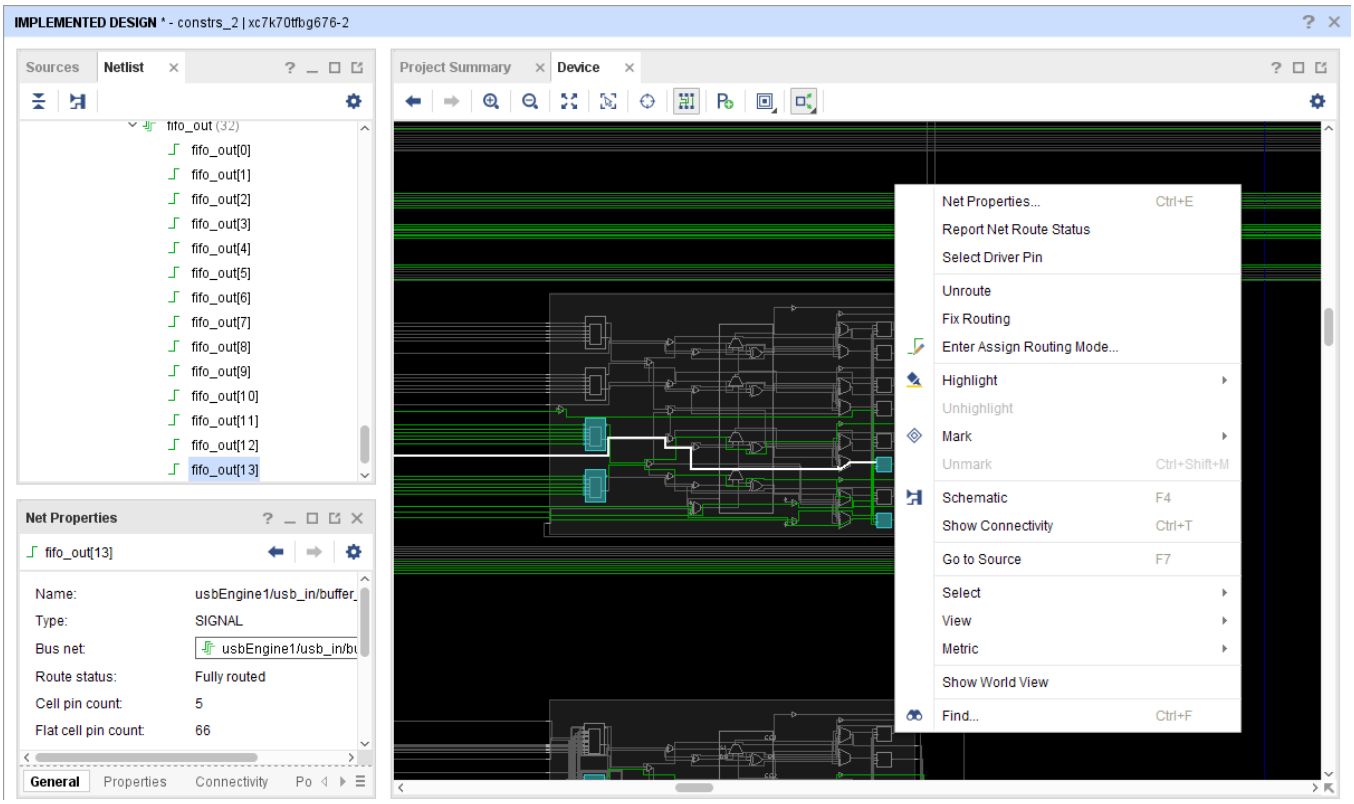
The Device View allows you to modify the routing for your design. You can Unroute, Route, and Fix Routing on any individual net.

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

1. Open Device window.
2. Select the net.
   - A red flyline indicates unrouted nets.
   - Partially routed nets are highlighted in yellow.
   - A dashed route. indicate nets with fixed routing.
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.
   - **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 47: **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. Assign LOC and BEL constraints to registers and I/Os, then manually route nets to control routing delay from the IOB to the register.

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:

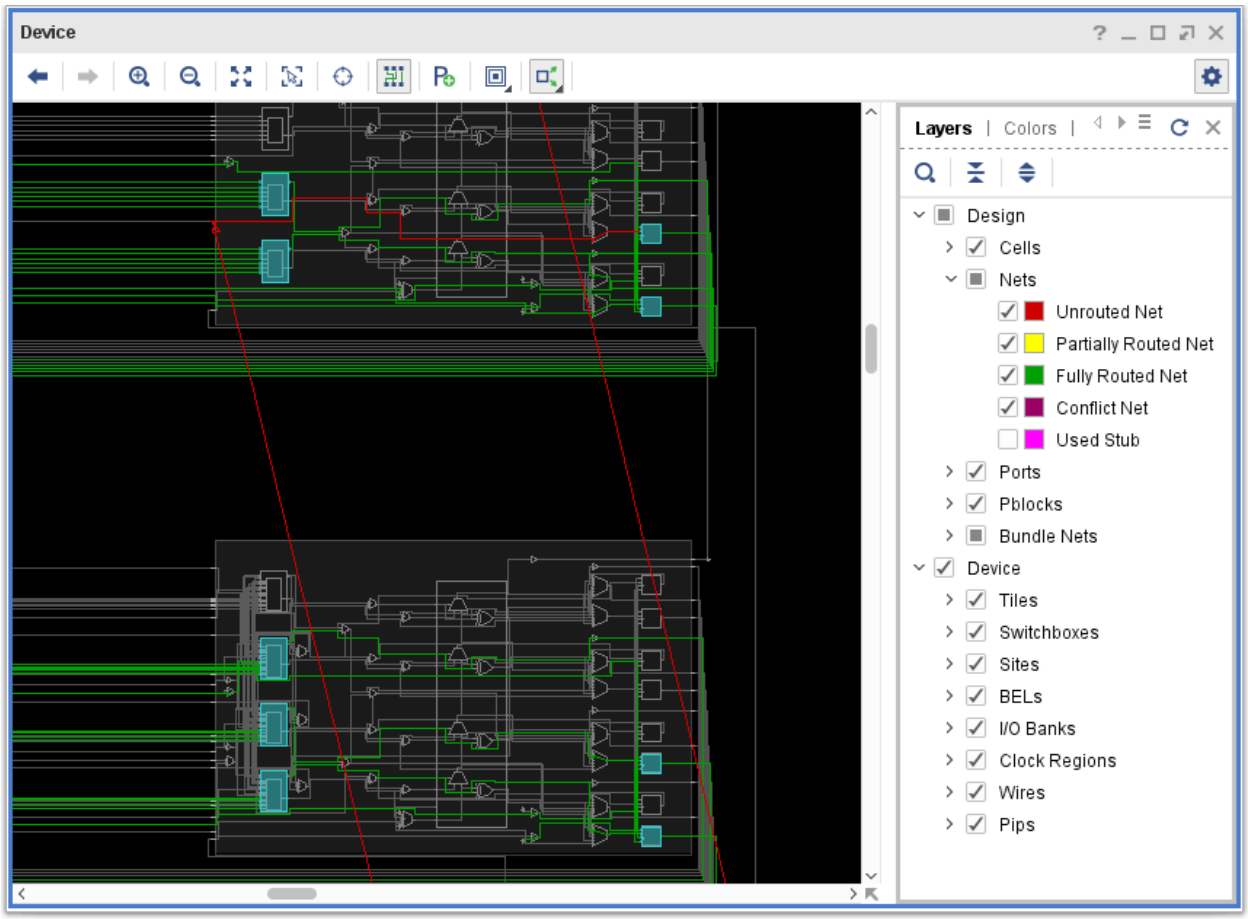
- The driver and the load require a LOC constraint and a BEL constraint.
- Branching is not allowed during manual routing, but you can implement branches by starting a new manual route from a branch point.

- 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 window.
2. Make sure to select **Routing Resources** in the Device window.
3. Enable the Layers for **Unrouted Net** and **Partially Routed Net** in the Device Options Layers view, shown in the following figure.



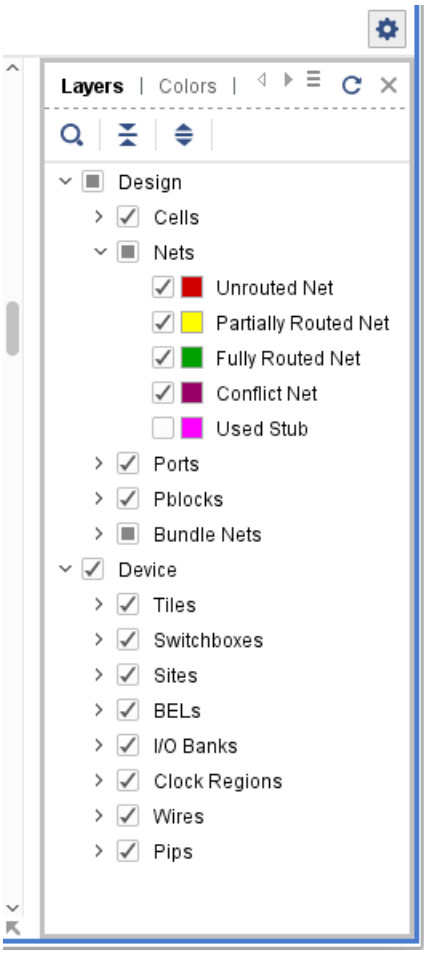
4. Select the net that requires routing.
   - Unrouted nets are indicated by a red flyline.
   - Partially routed nets are highlighted in yellow.

5. Right-click and select **Enter Assign Routing Mode**.

   The Assign Routing Mode: Target Load Cell Pin dialog box opens.

6. Optionally, select a load cell pin to which you want to route.
7. Click **OK**.

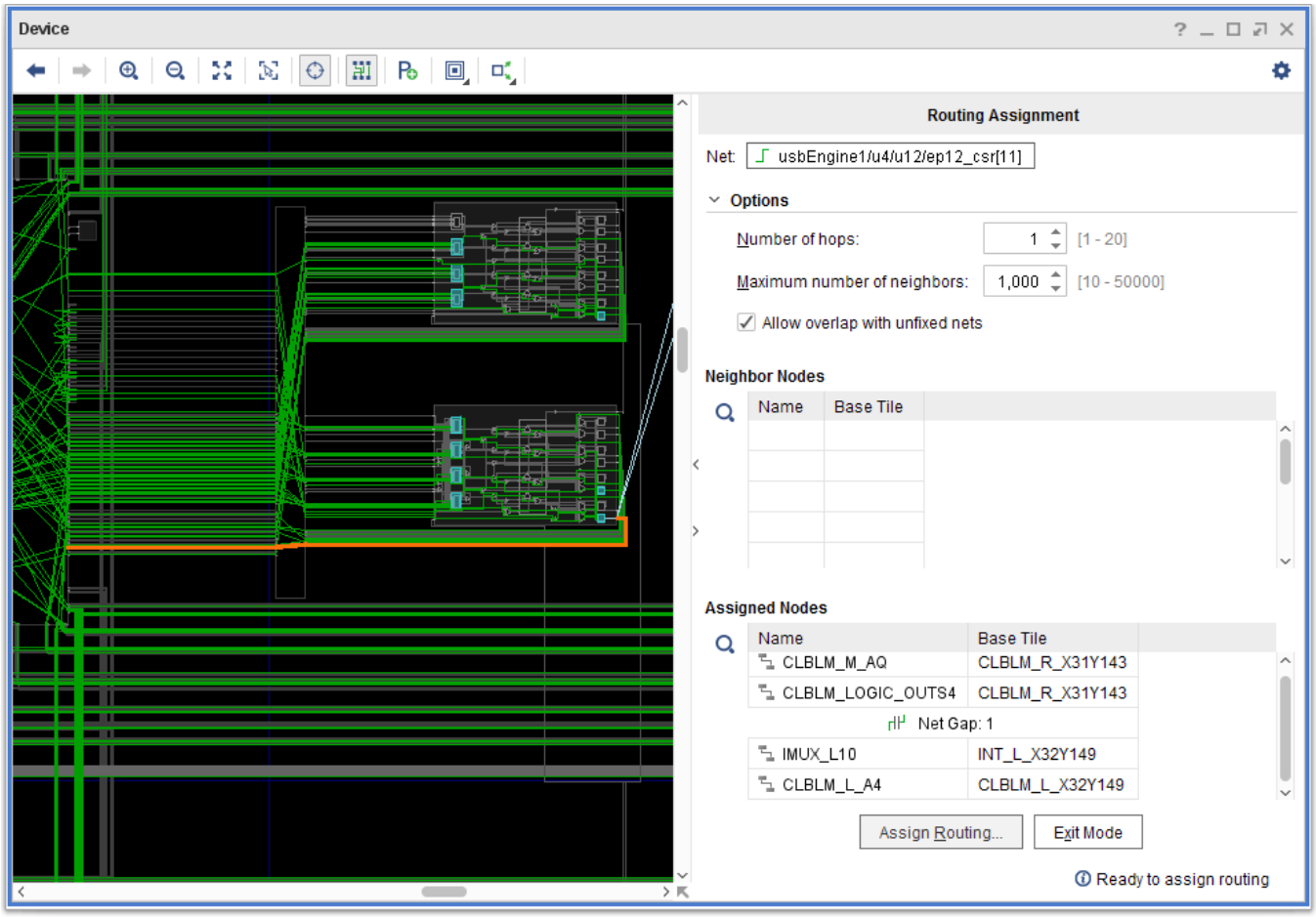
   ***Note*:** To display partially routed or unrouted nets in the Device window, ensure to select those layers in the Device Options menu.



You are now in Manual Routing Mode. A Routing Assignment window, shown in the following figure, appears next to the Device window.

Routing Assignment Window

Figure 48: **Routing Assignment Window**



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

- The Options section, shown in the following figure, controls the settings for the Routing Assignment window.

- 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 hop count is greater than 1, the Neighbor Nodes section shows only the last node in the route..
- The Maximum number of neighbors value limits the number of neighbor nodes 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**.

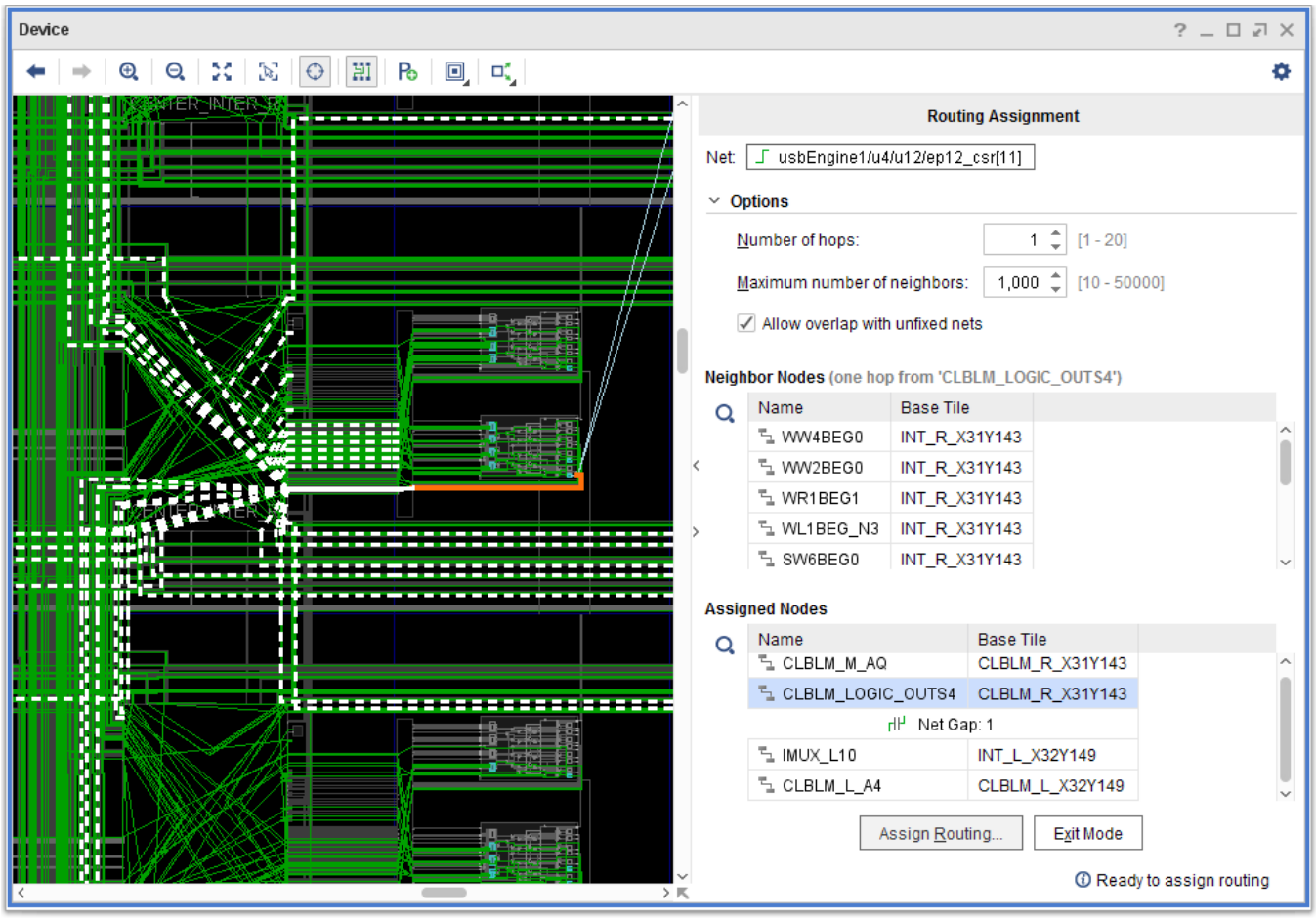
- The Assigned Nodes section shows the nodes that already have assigned routing. Each assigned node is displayed as a separate line item.

  In the Device window, nodes with assigned routing are highlighted in orange. Any gaps between assigned nodes appear in the Assigned Nodes section as a GAP line item. To auto-route gaps:

  - Right-click a net gap in the Assigned Nodes section.
  - 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 the following section) displays the allowed neighbor nodes, highlights the current selected nodes (in white). and highlights the allowed neighbor nodes (white dotted) in the Device window.

Figure 49: **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 assign 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**.

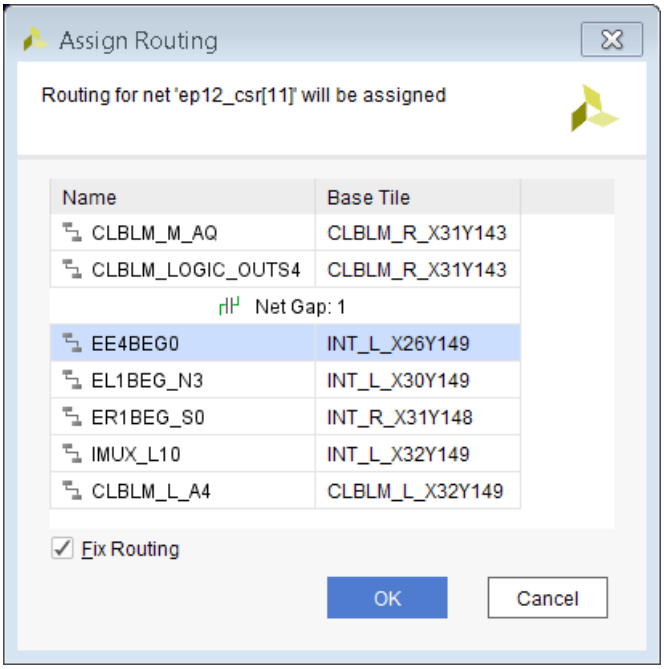
The nodes are removed from the assignment.

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 dialog box appears that allows to verify the assigned nodes before they are committed. See the following figure.

Figure 50: **Assign Routing Dialog Box**



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 window and select **Exit Assign Routing Mode**.

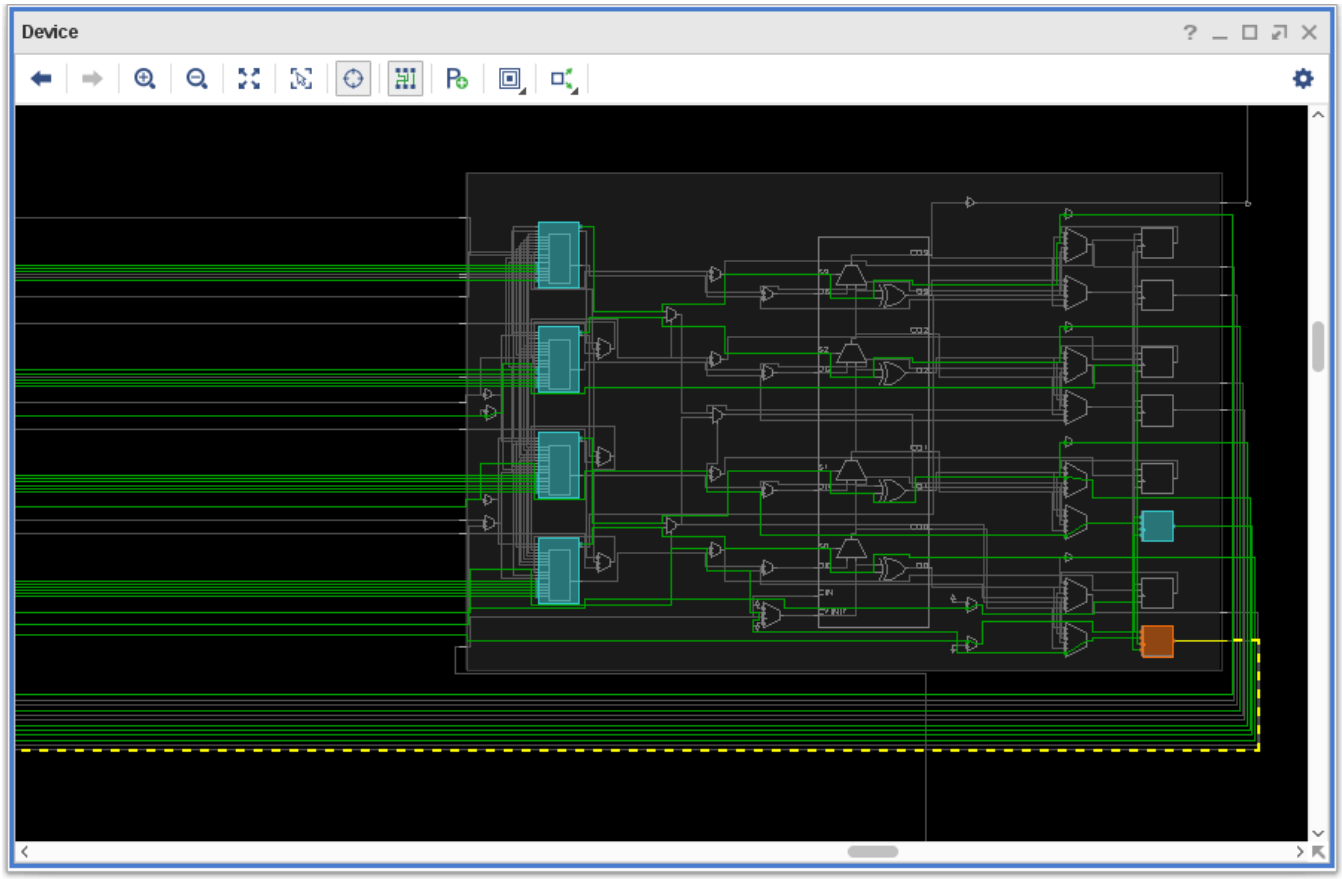
When the routes are committed, the driver and load BEL and LOC are also fixed.

Verifying Assigned Routes

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

The following figure shows an example of an assigned and partially assigned route.

Figure 51: **Assigned Partially Assigned Routing**



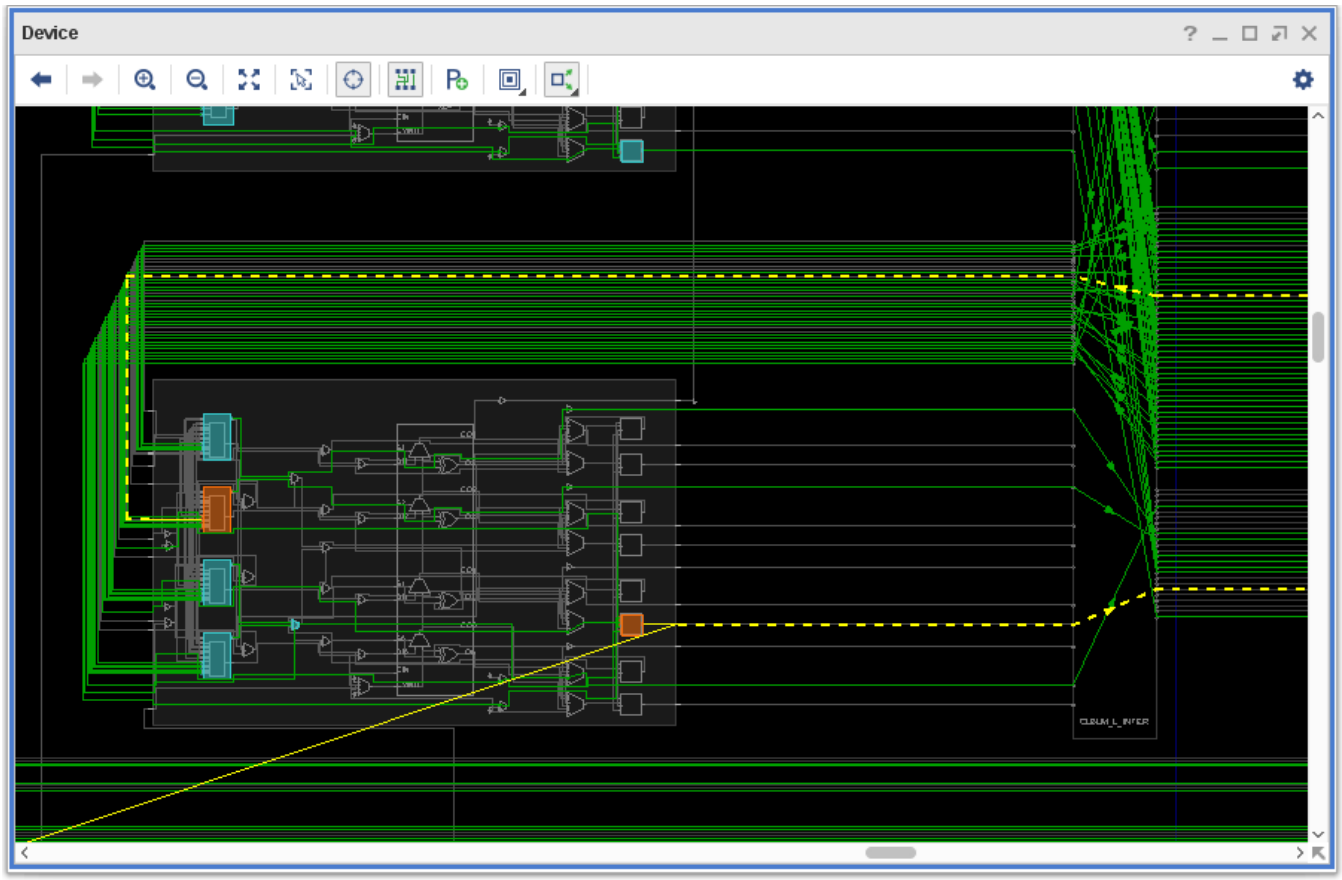
Branching

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.
2. Assign routing to all the branches of the net.

The following figure shows an example of a net that has assigned routing to one load and requires routing to two additional loads.

Figure 52: **Assign Branching Route**



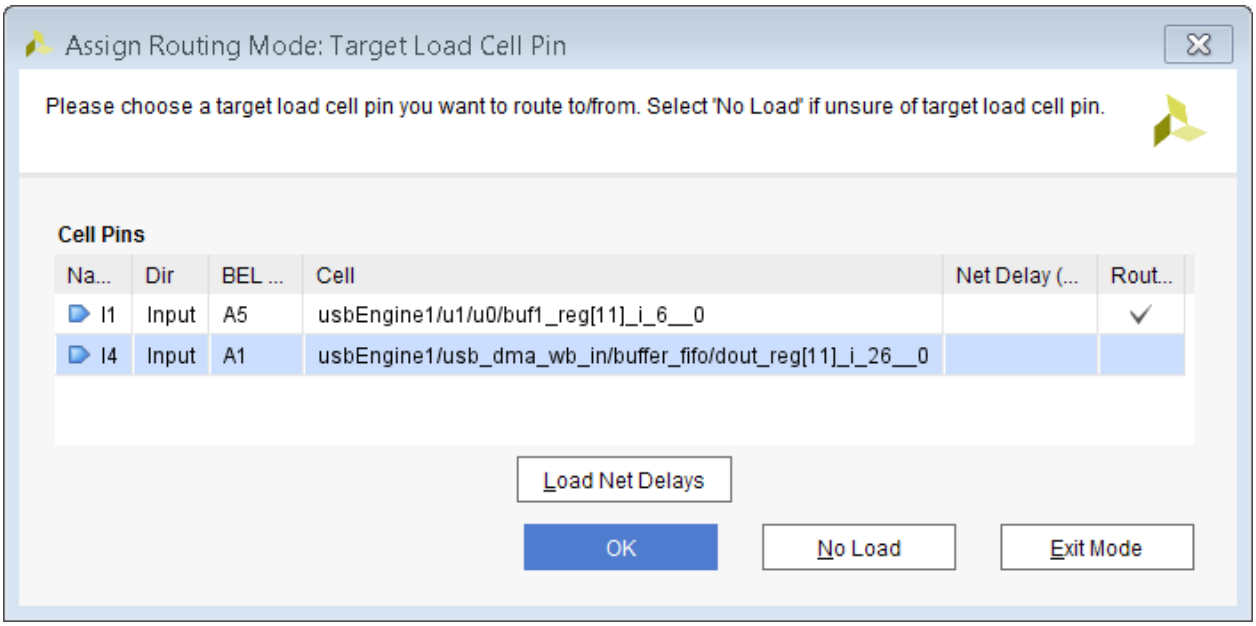
Assigning Routing to a Branch

To assign routing to a branch:

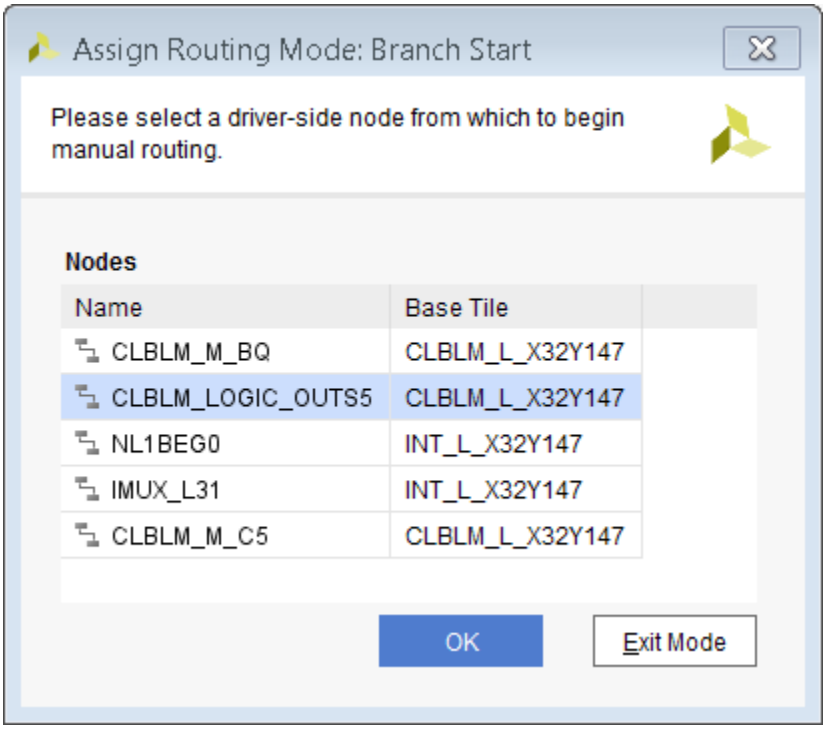
1. Go to Device window.
2. Select the net to be routed.
3. Right-click and select **Enter Assign Routing Mode**.

   The Assign Routing Mode: 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.



4. Select the load to which you want to route.
5. Click **OK**. The Assign Routing Mode: Branch Start dialog box, shown in the following figure, opens.



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.

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 window.
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, use the following Tcl script 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, nested {curly braces} indicates branching.

For example, consider the route described in the following figure. In this simplified illustration of a route, the various elements are indicated as shown in the following table (Directed Routing Constraints).

Table 21: **Directed Routing Constraints**

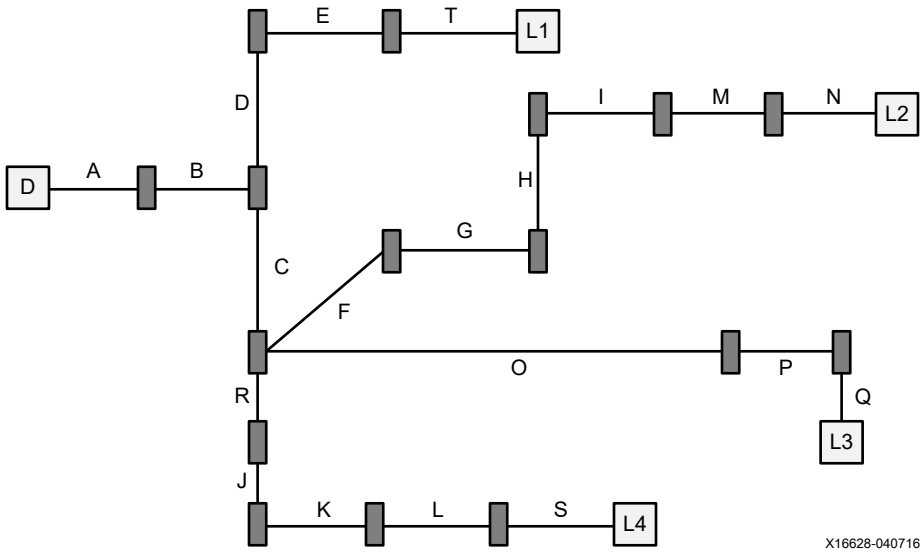
Elements	Indicated By
Driver and Loads	Orange Rectangles
Nodes	Red lines
Switchboxes	Blue rectangles

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 53: **Branch Route Example**



Using the find_routing_path Command to Create Directed Routing Constraints

You can use the `find_routing_path` Tcl command 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, you can find the nodes associated directly to the net. Refer to the *Vivado Design Suite Properties Reference Guide* (UG912) 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.

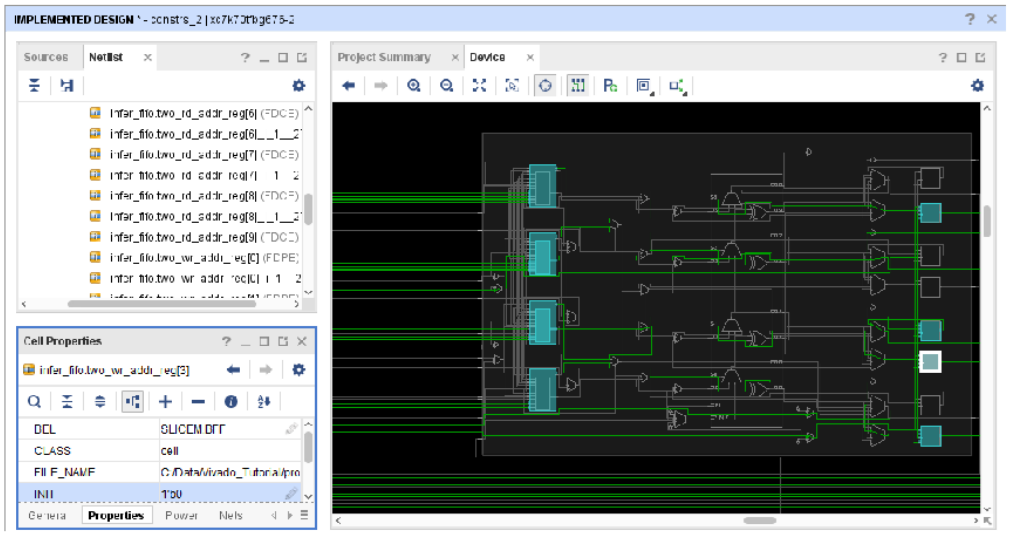
Modifying Logic

You can modify properties on logical objects that are not Read Only after Implementation in the Vivado IDE as well as Tcl.

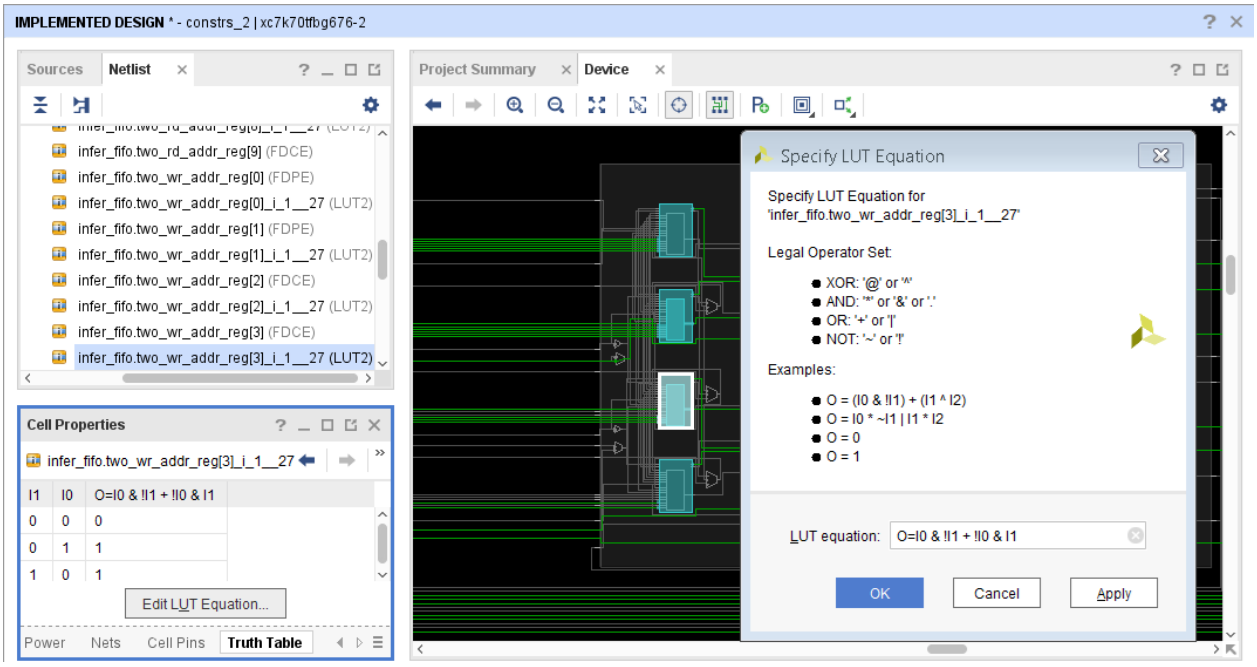
***Note*:** For more information about Tcl commands, see the *Vivado Design Suite Tcl Command Reference Guide* (UG835), or type `<command> -help`.

To modify a property on an object in Device window:

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



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.



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), or type `<command> -help`.

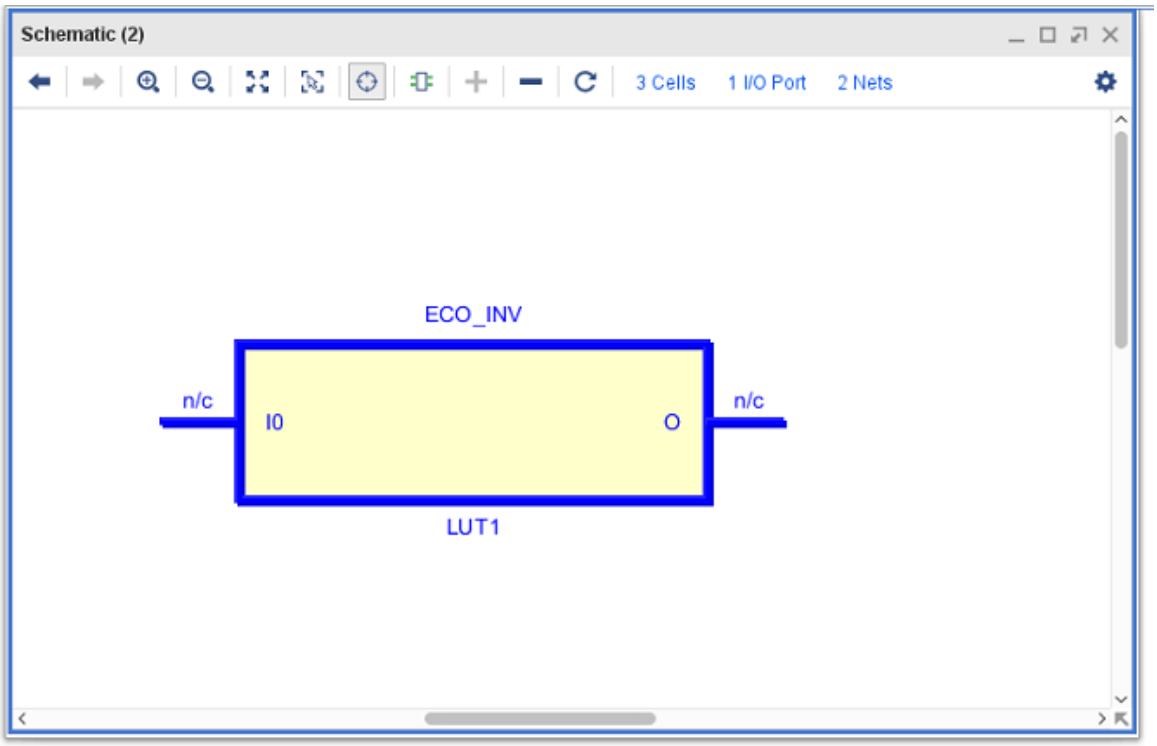
The netlist modifying commands work on a post-synthesis, post-place or post-route netlist. Before the netlist is modified, it must be loaded into memory. The netlist modifying commands allow you to make logical changes to the netlist when it is in memory. You can use the `write_checkpoint` command to 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. Run DRCs after performing your netlist changes.*

TIP: *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.*

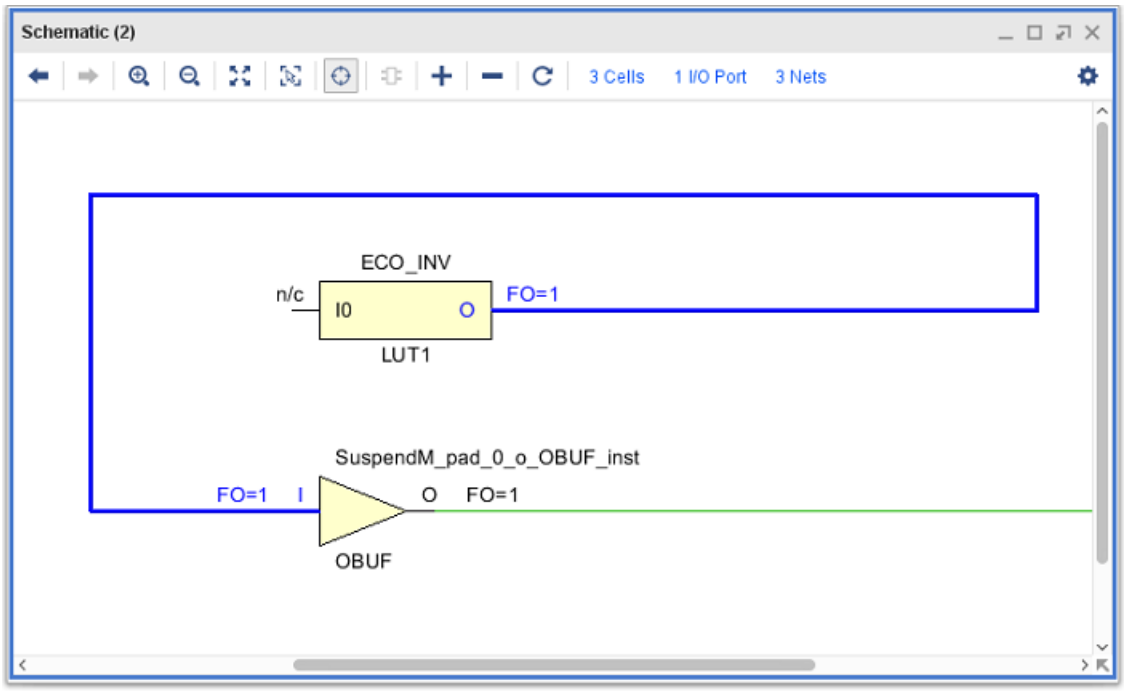
Logical changes reflect in the schematic view as soon as you execute the netlist modifying commands. The following figure shows an example of a cell created using a LUT1 as a reference cell.

Figure 54: **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." The following figure shows the resulting schematic view.

Figure 55: **Schematic After Connection of LUT1 to an OBUF**



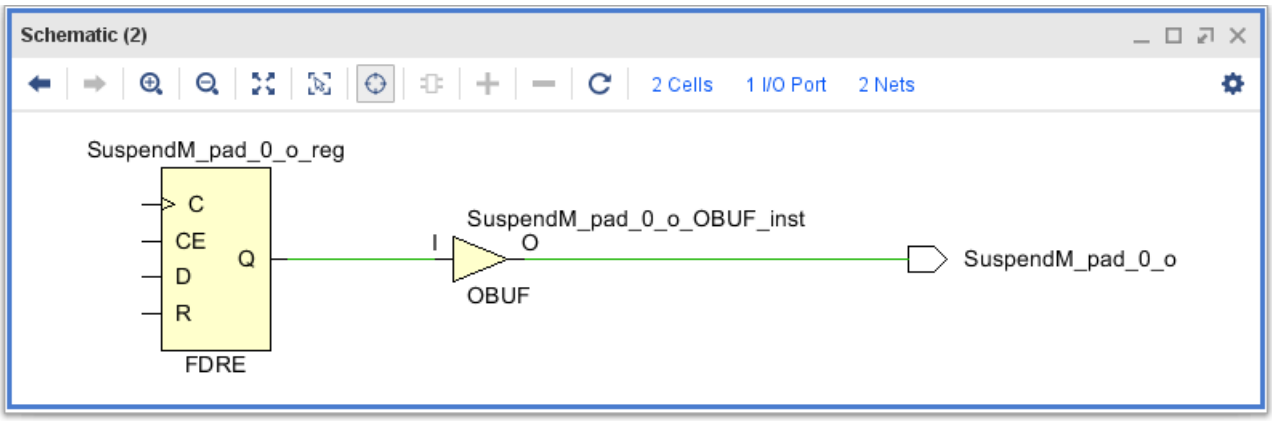
Use Cases

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 the following figure shows a FDRE primitive that is driving the output port `wbOutputData[0]` through an OBUF.

Figure 56: **FDRE Primitive Driving Output Port through an OBUF**



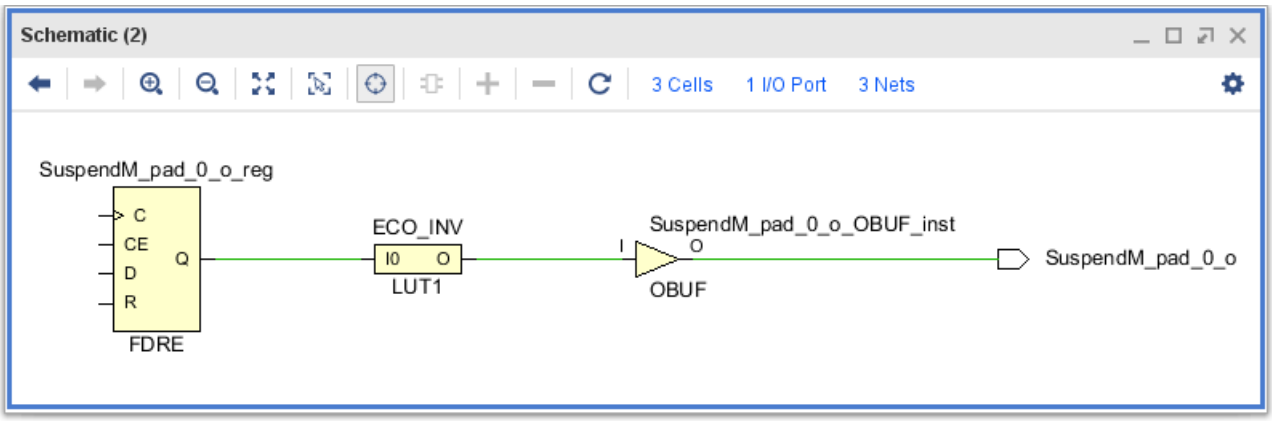
The following Tcl commands show how to add an inverter between the output of the FDRE and the OBUF:

```
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. 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.

The following figure shows the schematic of the resulting logical netlist changes.

Figure 57: **Schematic Showing Netlist Changes After Adding Inverter**



After you modify the netlist successfully, implement the logical changes. 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 you run `place_design` on the modified netlist. 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:

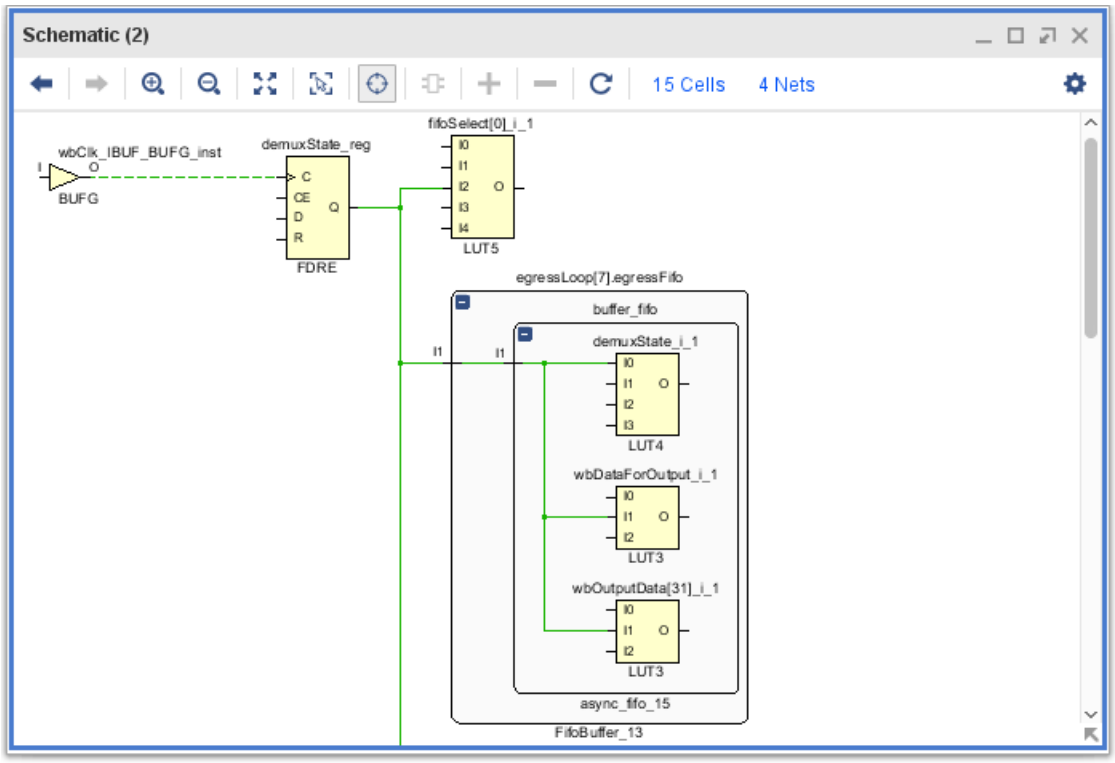
```
+-----------------------------------------------------+
|Incremental Routing Reuse Summary                    |
+-----------------------------------------------------+
|Type                  | Count    | Percentage     |
+----------------------+----------+----------------+
|Fully reused nets     |      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 58: **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.

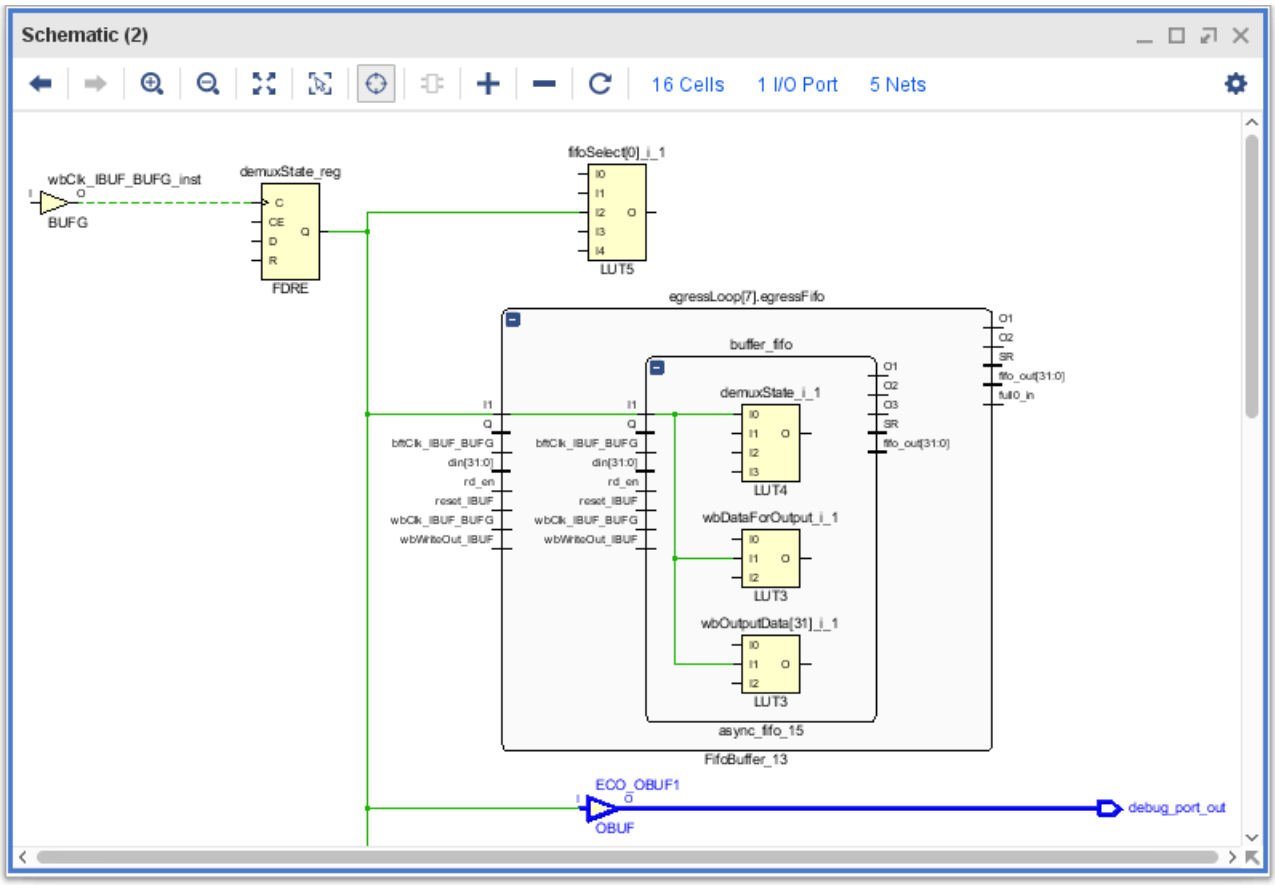
```
create_port -direction out debug_port_out
set_property PACKAGE_PIN AB20 [get_ports {debug_port_out}]
set_property IOSTANDARD LVCMOS18 [get_ports [list debug_port_out]]
create_cell -reference [get_lib_cells [get_libs]/OBUF] ECO_OBUF1
create_net ECO_OBUF1_out
connect_net -net ECO_OBUF1_out -objects 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`.

The following figure shows the schematic of the resulting logical netlist changes.

Figure 59: **Schematic after Adding/Routing a Debug Port**



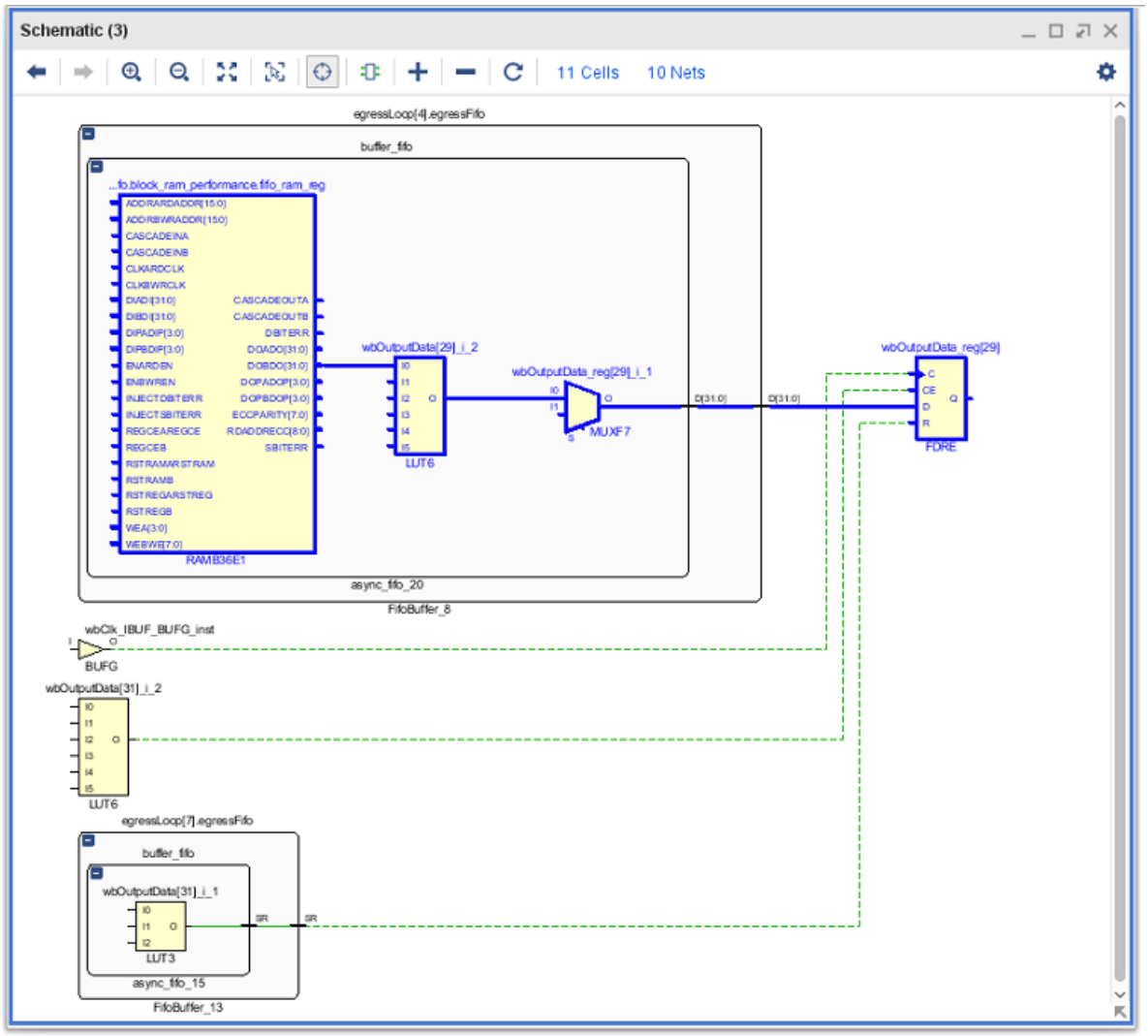
After modifying the netlist successfully, implement the logical changes. Because the port has been assigned to a package pin, the OBUF driving the port is automatically placed in the correct location. Therefore, the placer does not have anything to place and therefore incremental compile is not triggered 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 can change the routing for other nets. Alternatively, you can run `route_design -preserve`, which preserves existing routing. See Using Other route_design Options.

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 the following figure 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 60: **Schematic Prior to Addition of Pipeline Register**



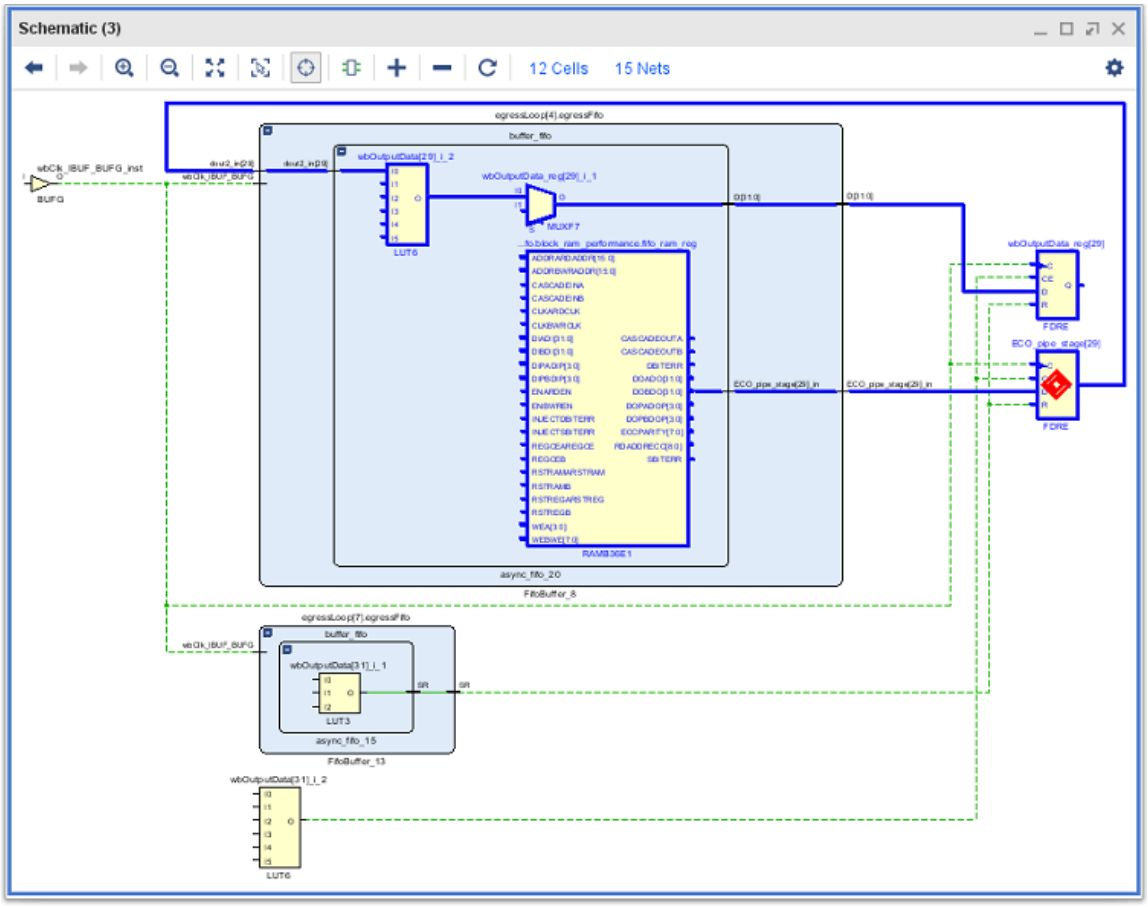
The following Tcl script shows how to insert a pipeline register between the two LUT6 cells. The register is implemented with the same control signals as the load register.

```
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 following figure shows the schematic of the resulting logical netlist changes.

Figure 61: **Schematic Showing Addition of Pipeline Register**



Commit logical changes after modifying the netlist successfully. Accomplish this using the `place_design` and `route_design` commands.

Vivado ECO Flow

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 do the following:

1. Modify a design checkpoint
2. Implement the changes
3. Run reports on the changed netlist
4. Generate programming files

Common use cases for the ECO flow are:

- Modifying debug probes of ILA and VIO cores in the design.
- Routing an internal net to a package pin for external probing.
- Evaluating what-if scenarios such as improving timing and fixing functional bugs.

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**.

IMPORTANT! *ECOs only work on design checkpoints. The ECO Layout is only available after a design checkpoint is open in the Vivado IDE.*

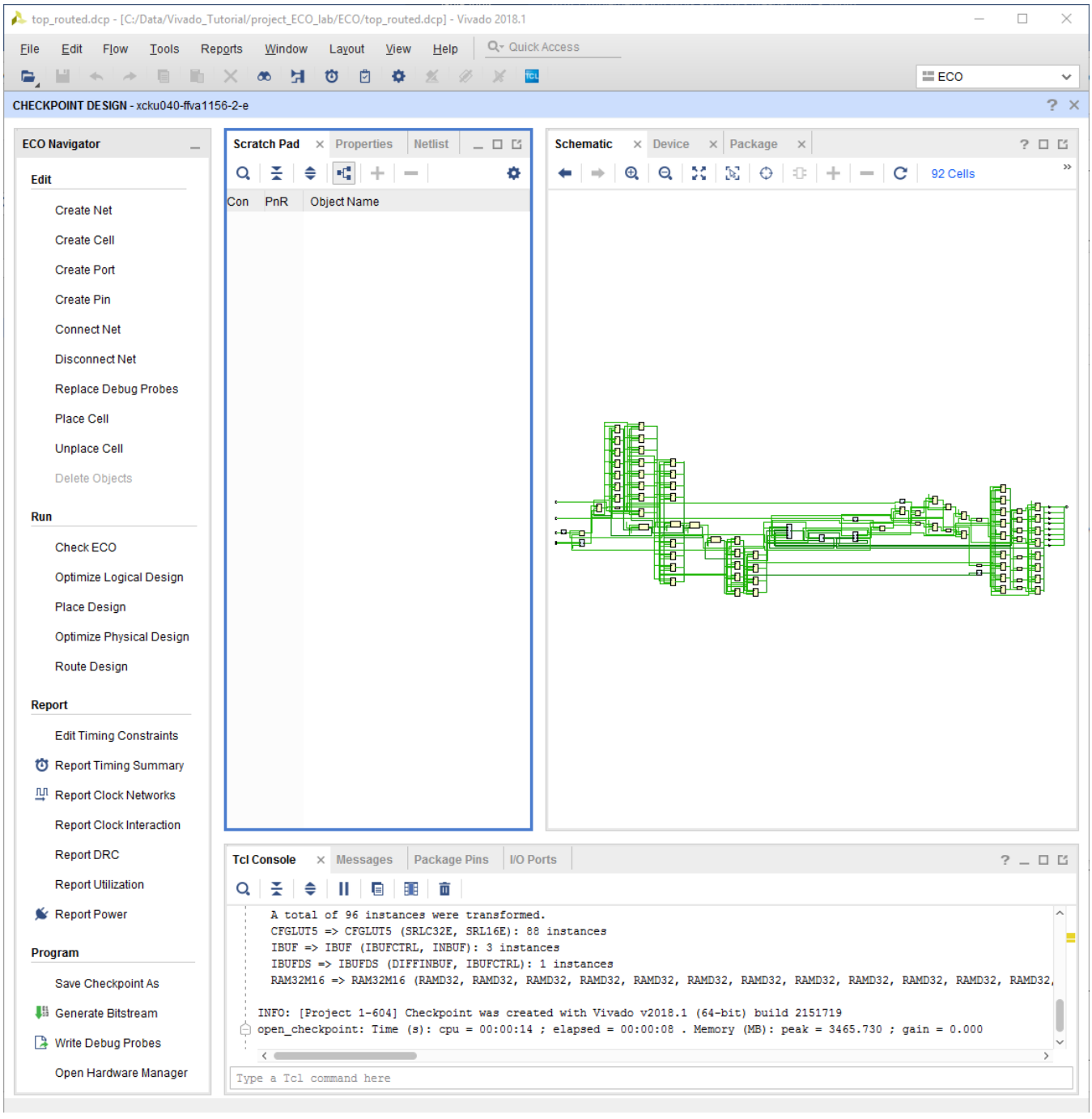
ECO Navigator

The ECO Navigator provides access to the commands 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 62: **Vivado ECO Layout**

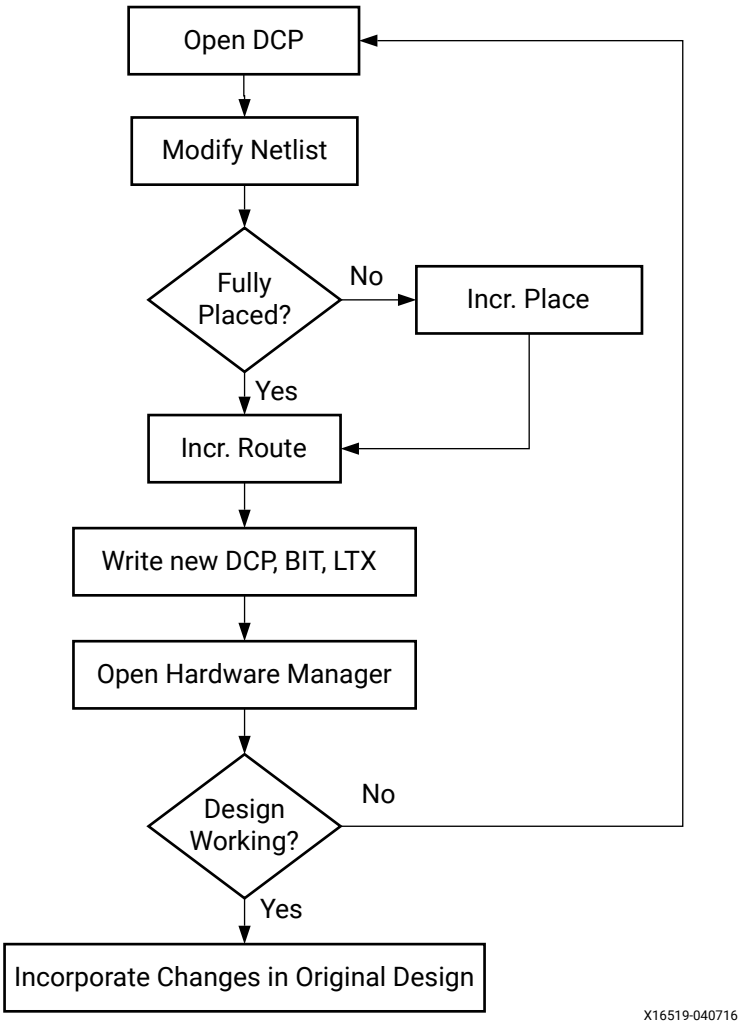


ECO Flow Chart

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. Next, 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 63: **ECO Flow Chart**



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

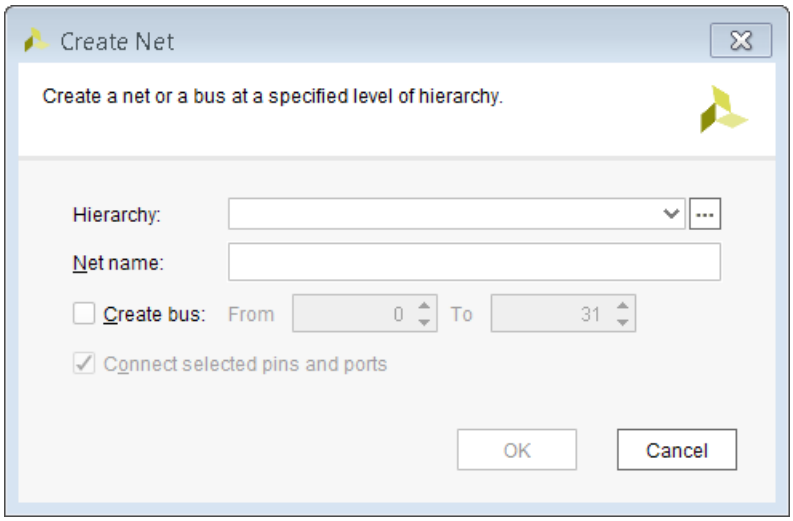
The Edit section of the ECO Navigator (shown in the following figure) provides access to all the commands that are required to modify the netlist.

Figure 64: **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, select the **Connect selected pins and ports** check box to automatically connect the new net to them.

Figure 65: **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. You can also 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 66: **Create Cell Dialog Box**

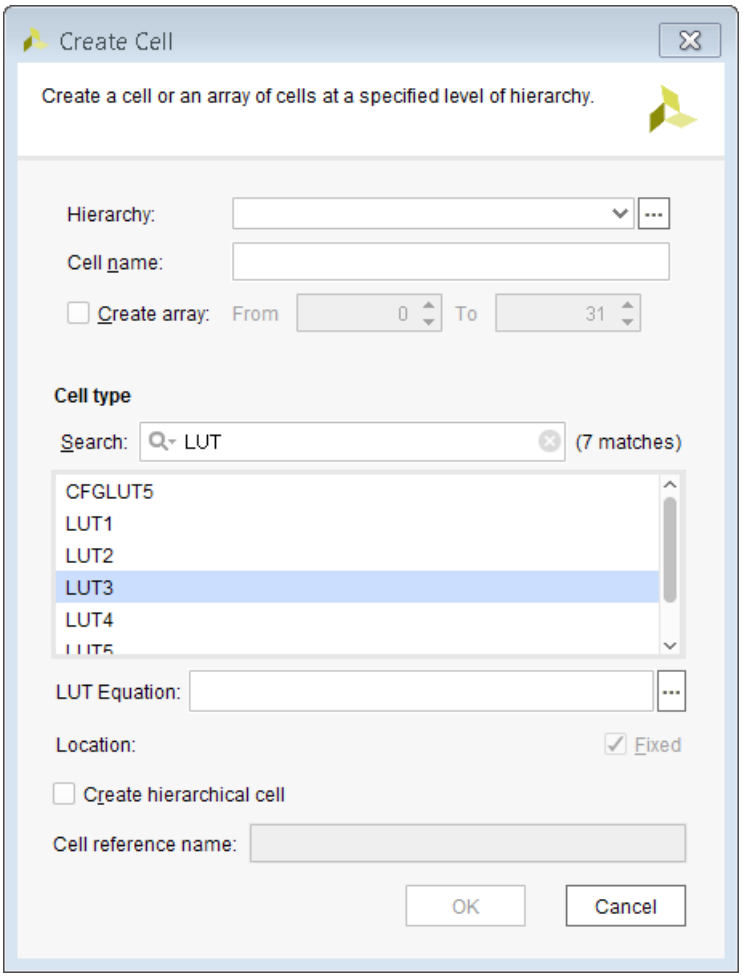
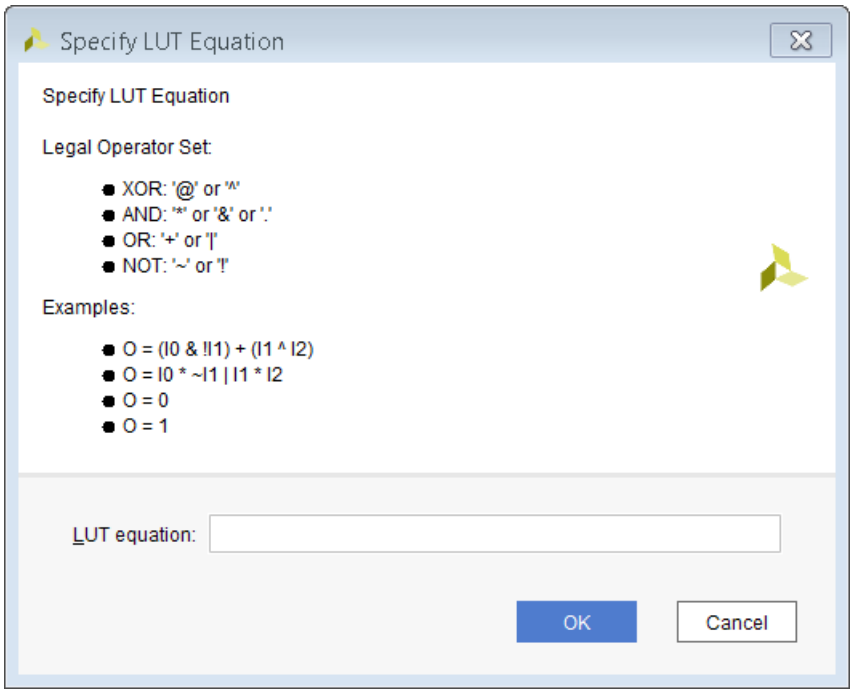
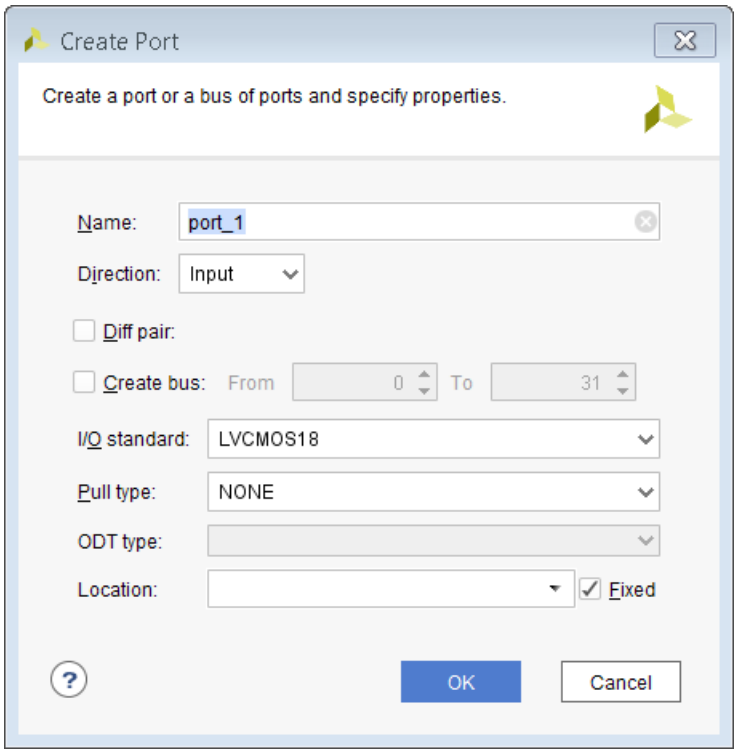


Figure 67: **Specify LUT Equation Dialog Box**



- **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.

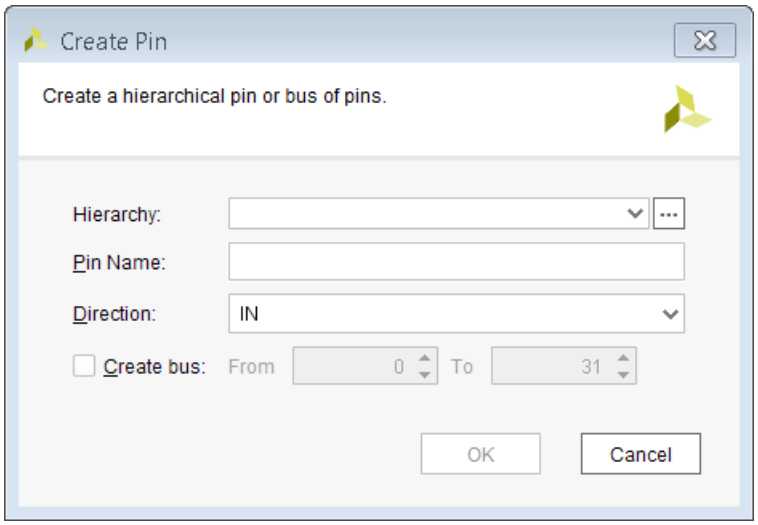
Figure 68: **Create Port Dialog Box**



- **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.

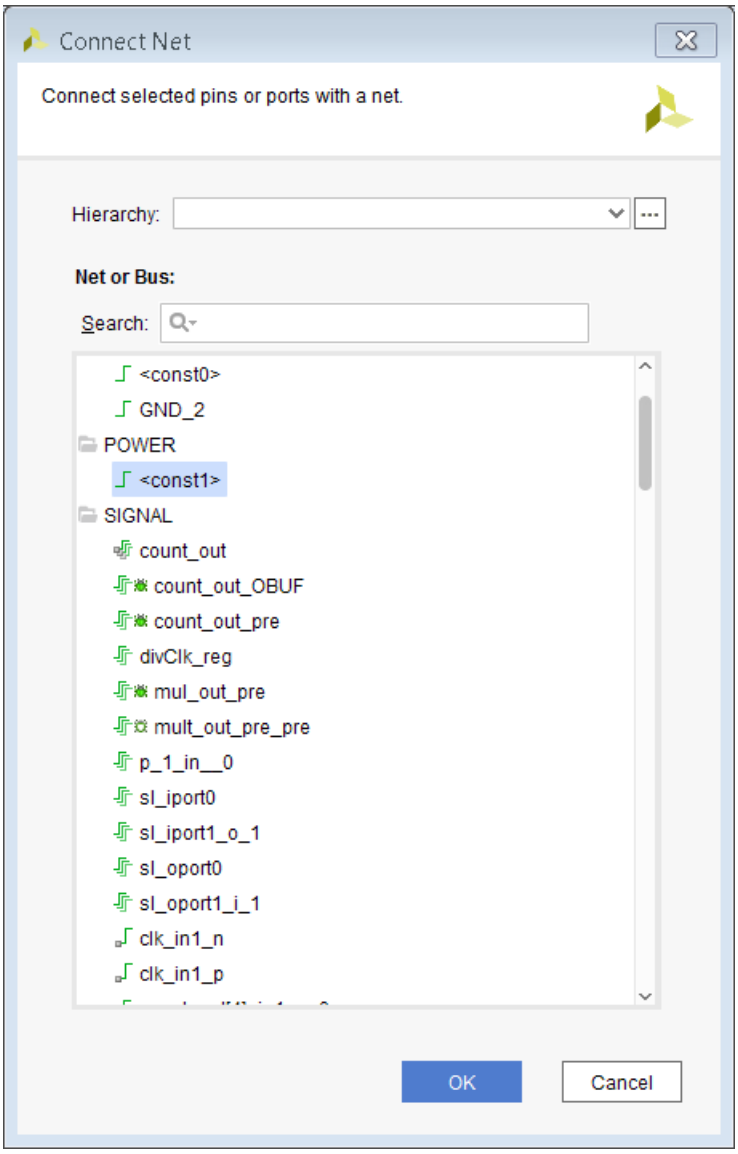
  You can create bus pins with increasing or decreasing bus indexes, using negative and positive index values. Create a pin on an existing cell instance. Otherwise, it is considered a top-level pin, which should be created using the `create_port` command. You cannot create a pin if you do not specify the instance name of a cell.

Figure 69: **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. It 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 is connected across levels of hierarchy in the design, by adding pins and hierarchical nets as needed to complete the connection.

Figure 70: **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 are 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 71: **Replace Debug Probes Dialog Box**

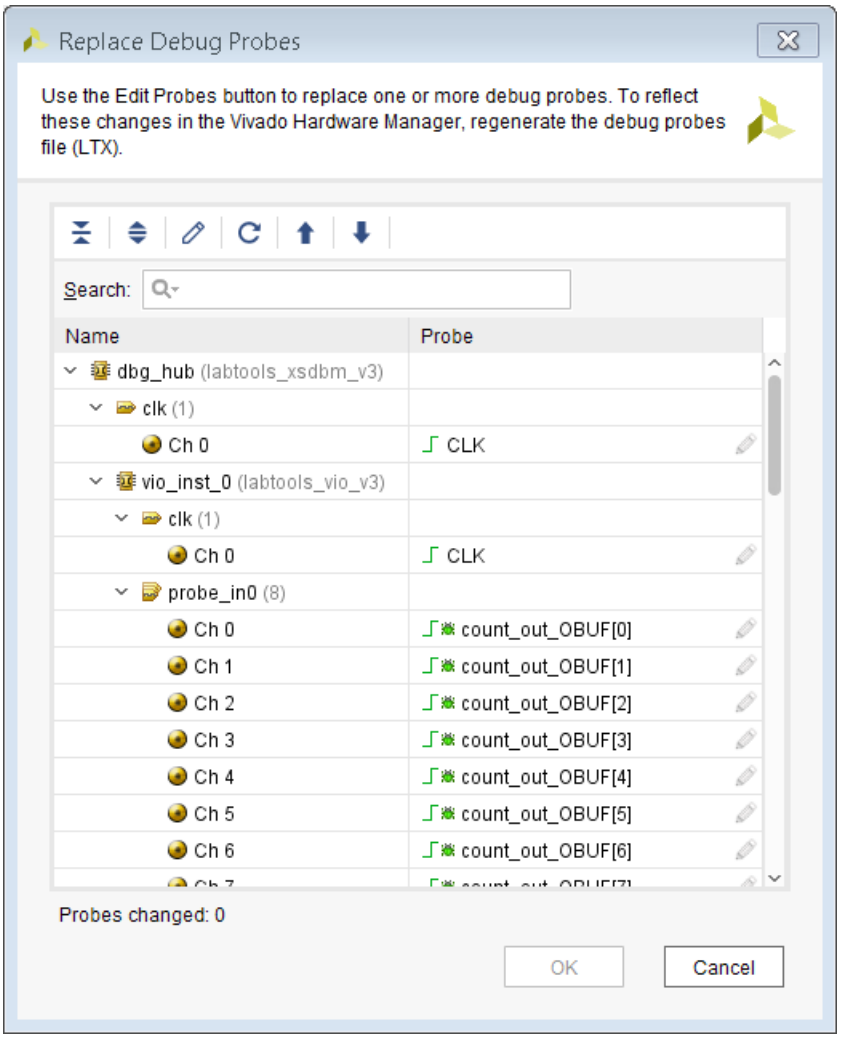
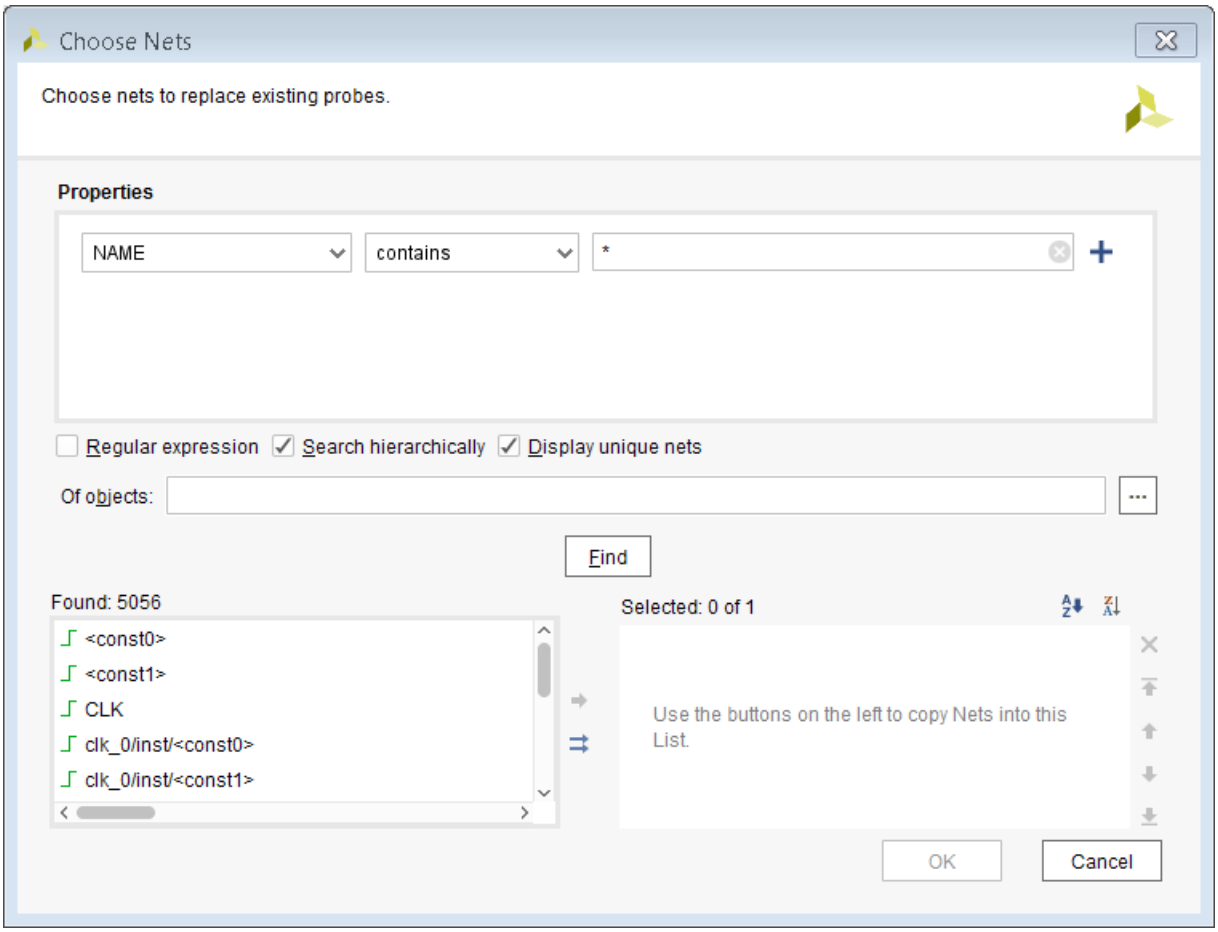


Figure 72: **Choose Nets Dialog Box**

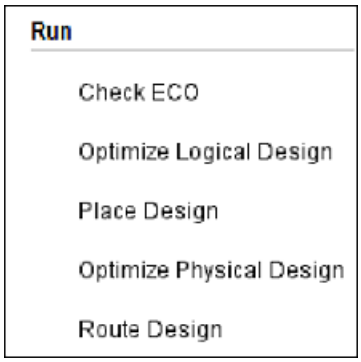


- **Place Cell:** Places the selected cell onto the selected device resource.
- **Unplace Cell:** Unplaces the selected cell from its current placement site.
- **Delete Objects:** Deletes the selected objects from the current design.

Run Section

The Run section of the ECO Navigator, shown in the following figure, provides access to all the commands required to implement the current changes.

Figure 73: **ECO Navigator Run Commands**

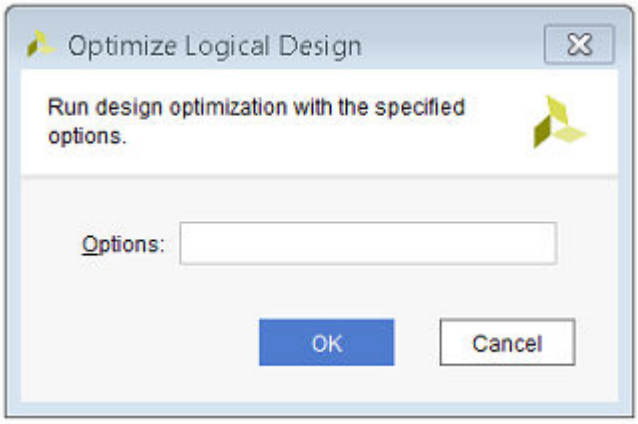


- **Check ECO:** Runs the ECO checks rule deck on the current design.

> **TIP:** *The Vivado tools allow 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 74: **Optimize Logical Design Dialog Box**



- **Place Design:** For 7 series and UltraScale families, 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 for additional information on Incremental Place and Route.

For Versal, `place_design -eco` command invokes the ECO placer. By default, it supports timing driven mode. To invoke the non-timing driven mode, an extra switch `-no_timing_driven` is used. The ECO placer attempts to locate newly added cells close to their connections and in some cases may need to move placed cells aside to accommodate them. The addition of new cells is limited to 1% of the design utilization and to the addition of Flops and LUTs. For DFX designs, ECO edits are allowed in child implementation within the reconfigurable module. Static modifications are not supported.

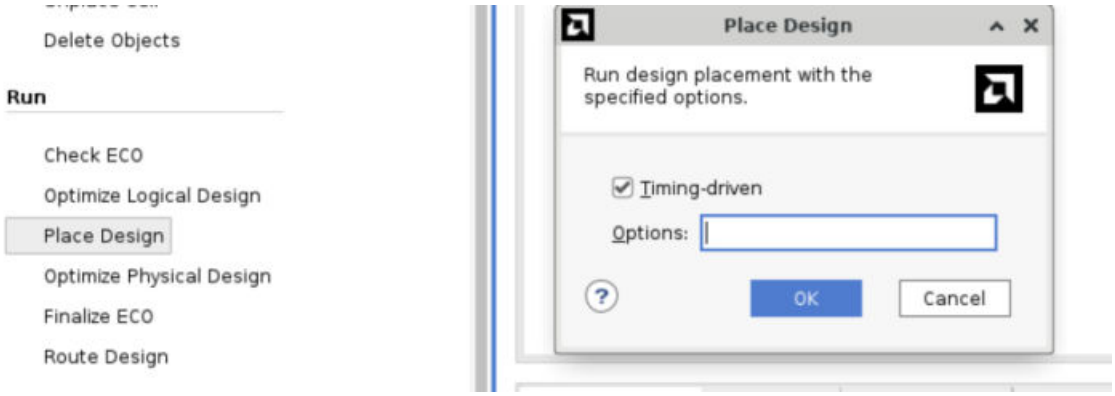
The following snapshot shows how to trigger ECO Placer in the GUI mode. To run in non-timing driven mode, uncheck the timing-driven mode box. The Tcl command to invoke ECO placer is as follows:

```
place_design -eco
```

For non timing driven mode use:

```
place_design -eco -no_timing_driven
```

Figure 75: **Place Design Dialog Box**



To better understand the impact of the placer, see the ECO Placement Report available from the `vivado.log` file.ECO Placement Report

Figure 76: **ECO Placement Report**

ECO Placement Report

Cells	Number	Percentage
# Logical Cells	7373102	100.00
Placed	7373102	100.00
Unplaced	0	0.00
ECO Placed	94	0.00
ECO Replaced	0	0.00
ECO Preserved	7147659	96.94

The entries in the preceding table are as follows:

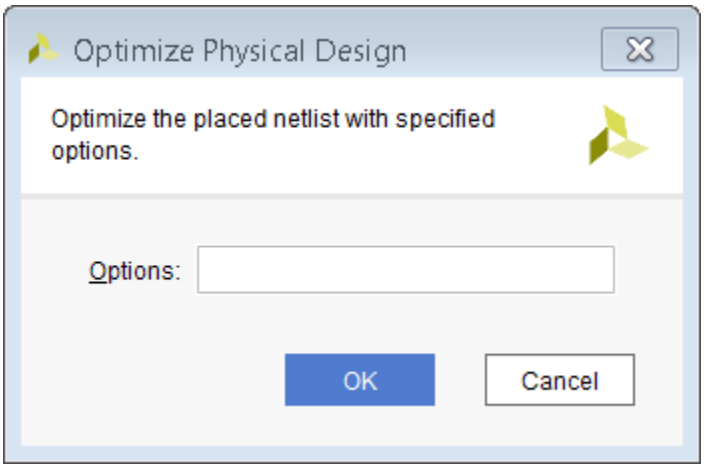
- **Logical Cells:** Total number of cells in the design. Logical cells = Placed + Unplaced
- **Placed:** Number of instances placed at the completion of the ECO Placer (including the preserved instances)
- **Unplaced:** Number of instances unplaced at the completion of the ECO Placer
- **ECO Placed:** Number of unplaced cells that got placed by the ECO Placer
- **ECO Replaced :** Number of originally placed cells that are replaced by the ECO Placer
- **ECO Preserved :** Number of cells that are not touched by the ECO Placer

The ECO placer generates a separate text file, `eco_placer_modified_cellInfo.txt` to capture the original cell movement referenced in the ECO Replaced entry of the ECO Placement Report. The snapshot of the text file is as follows:

Cell Number	Type	Name	From LOC/BEL	To LOC/BEL	Combined
1	FDRE	fm_cset_rewrite_data_out[37]	SLICE_S0X220Y169/GFF	SLICE_S0X220Y169/GFF2	0
2	FDRE	fm_cset_rewrite_data_out[37]	SLICE_S0X220Y169/GFF2	SLICE_S0X220Y169/HFF	0
3	FDRE	fm_cset_rewrite_q[7]	SLICE_S0X198Y223/DFF	SLICE_S0X198Y223/DFF2	0
4	FDRE	fm_cset_rewrite_q[6]	SLICE_S0X198Y223/DFF2	SLICE_S0X198Y223/EFF	0
5	LUT3	sqnod_payload_dst[47]	SLICE_S0X198Y223/E5LUT	SLICE_S0X198Y223/E6LUT	1
6	LUT3	sqnod_payload_dst[46]	SLICE_S0X198Y223/E6LUT	SLICE_S0X198Y223/F5LUT	1
7	FDRE	fm_cset_rewrite_q[9]	SLICE_S0X198Y223/EFF	SLICE_S0X198Y223/EFF2	0
8	FDRE	fm_cset_rewrite_q[8]	SLICE_S0X198Y223/EFF2	SLICE_S0X198Y223/FFF	0
9	LUT3	sqnod_payload_dst[49]	SLICE_S0X198Y223/F5LUT	SLICE_S0X198Y223/F6LUT	1
10	LUT3	sqnod_payload_dst[48]	SLICE_S0X198Y223/F6LUT	SLICE_S0X198Y223/G5LUT	1
11	FDRE	fm_cset_rewrite_q[11]	SLICE_S0X198Y223/FFF	SLICE_S0X198Y223/FFF2	0
12	FDRE	fm_cset_rewrite_q[10]	SLICE_S0X198Y223/FFF2	SLICE_S0X198Y223/GFF	0

- **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 77: **Optimize Physical Design Dialog Box**



Refer to Incremental Implementation 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.

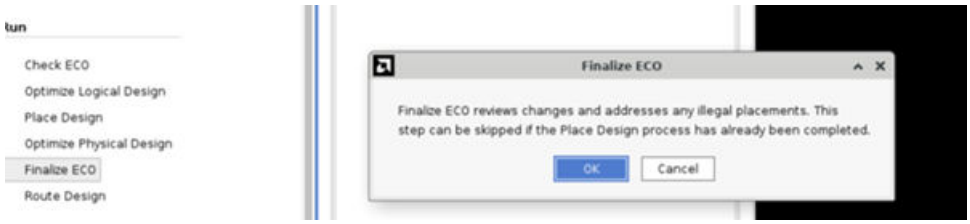
For Versal, `route_design -eco` command invokes the ECO router flow. It supports both timing driven and non timing driven modes. Use the `finalize_eco` command to ensure valid placement before calling the ECO router flow. Invalid site configurations resulting from ECO changes are fixed by `finalize_eco`. It also incrementally updates the clock network if the movement of cells is within the clock expansion window. It generates a separate text file, `finalize_eco_modified_cellInfo.txt` to capture the cells moved by `finalize_eco` to fix illegal site configurations.

Cell Number	Type	Name	From LOC/BEL	To LOC/BEL	Combined
1	FDRE	fm_cset_rewrite_data_out[37]	SLICE_S0X220Y169/GFF	SLICE_S0X220Y169/GFF2	0
2	FDRE	fm_cset_rewrite_data_out[37]	SLICE_S0X220Y169/GFF2	SLICE_S0X220Y169/HFF	0
3	FDRE	fm_cset_rewrite_q[7]	SLICE_S0X198Y223/DFF	SLICE_S0X198Y223/DFF2	0
4	FDRE	fm_cset_rewrite_q[6]	SLICE_S0X198Y223/DFF2	SLICE_S0X198Y223/EFF	0
5	LUT3	sqnod_payload_dst[47]	SLICE_S0X198Y223/E5LUT	SLICE_S0X198Y223/E6LUT	1
6	LUT3	sqnod_payload_dst[46]	SLICE_S0X198Y223/E6LUT	SLICE_S0X198Y223/F5LUT	1
7	FDRE	fm_cset_rewrite_q[9]	SLICE_S0X198Y223/EFF	SLICE_S0X198Y223/EFF2	0
8	FDRE	fm_cset_rewrite_q[8]	SLICE_S0X198Y223/EFF2	SLICE_S0X198Y223/FFF	0
9	LUT3	sqnod_payload_dst[49]	SLICE_S0X198Y223/F5LUT	SLICE_S0X198Y223/F6LUT	1
10	LUT3	sqnod_payload_dst[48]	SLICE_S0X198Y223/F6LUT	SLICE_S0X198Y223/G5LUT	1
11	FDRE	fm_cset_rewrite_q[11]	SLICE_S0X198Y223/FFF	SLICE_S0X198Y223/FFF2	0
12	FDRE	fm_cset_rewrite_q[10]	SLICE_S0X198Y223/FFF2	SLICE_S0X198Y223/GFF	0

If the ECO operation adds new cells, `finalize_eco` is not required and you can run `place_design -eco` to place the cells before routing. Otherwise, run `finalize_eco` before `route_design -eco`.

In the GUI, when you click `finalize_eco`, a dialog appears to verify whether `place_design -eco` has already run. Click **OK** to continue the `finalize_eco` flow.

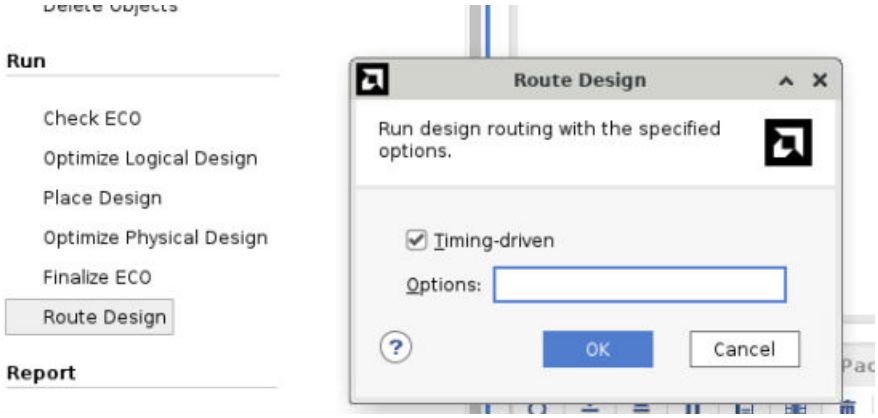
Figure 78: **Finalize Eco Dialog Box**



The following figure shows how to trigger the ECO Router in the GUI. To run in non–timing-driven mode, clear the Timing-driven mode checkbox. Invoke the ECO Router with the following Tcl command:

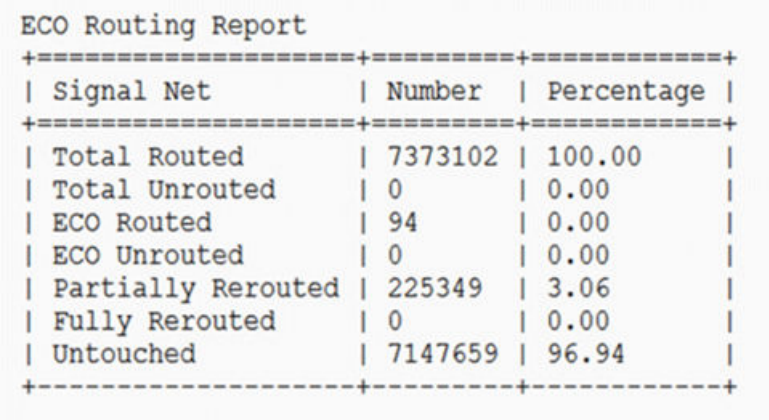
- route_design `-eco` for timing driven flow
- route_design `-eco -no_timing_driven` for non-timing driven flow

Figure 79: **Route Design Dialog Box**



To better understand the impact of the router, see the ECO Routing Report available from the `vivado.log` file.

Figure 80: **ECO Routing Report**



ECO Routing Report

Signal Net	Number	Percentage
Total Routed	7373102	100.00
Total Unrouted	0	0.00
ECO Routed	94	0.00
ECO Unrouted	0	0.00
Partially Rerouted	225349	3.06
Fully Rerouted	0	0.00
Untouched	7147659	96.94

The entries in the preceding table are described as follows:

- **Total Routed:** Total number of nets routed in the design, including both original and ECO-modified nets
- **Total Unrouted:** Number of nets unrouted in the design.
- **ECO Routed:** Number of nets routed by ECO router that were added or modified through ECO operations.
- **ECO Unrouted:** Number of nets that failed to route by the ECO router that were added or modified through ECO operations.
- **Partially Rerouted:** Number of nets that were not involved in ECO operations, which were partially impacted when finding a legal solution.
- **Fully Rerouted:** Number of nets that were not involved in ECO operations, which were fully impacted when finding a legal solution.
- **Untouched:** Number of reused nets from the previous routing solution.

After running the ECO flow, store the changes to a new checkpoint, write new programming and debug probe files, and open the **Hardware Manager** to program your device.

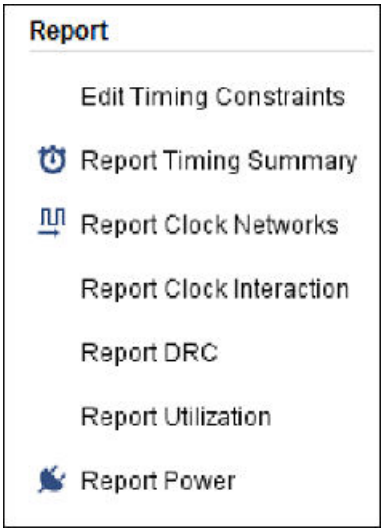
ECO Flow Analysis

Use the following reports to analyze the ECO flow. Run `report_place_status` after ECO edits to check for unplaced cells and illegal site configurations, then run `finalize_eco` to fix any violations. Use `report_route_status` to assess routing after ECO edits and to see the number of unrouted or affected nets. If `report_place_status` reports no unplaced cells and no illegal site configurations, run the ECO Router to route the affected nets. For more details on these reports, see *Vivado Design Suite Tcl Command Reference Guide* (UG835)

Report Section

The Report Section of the ECO Navigator provides access to all the commands required to run reports on the modified design.

Figure 81: **ECO Navigator Report Commands**

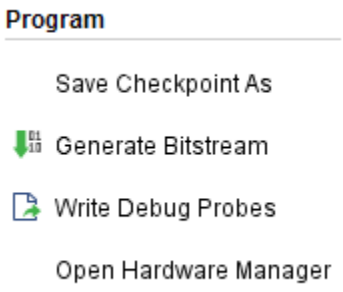


For more information on these commands, refer to the *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893).

Program Section

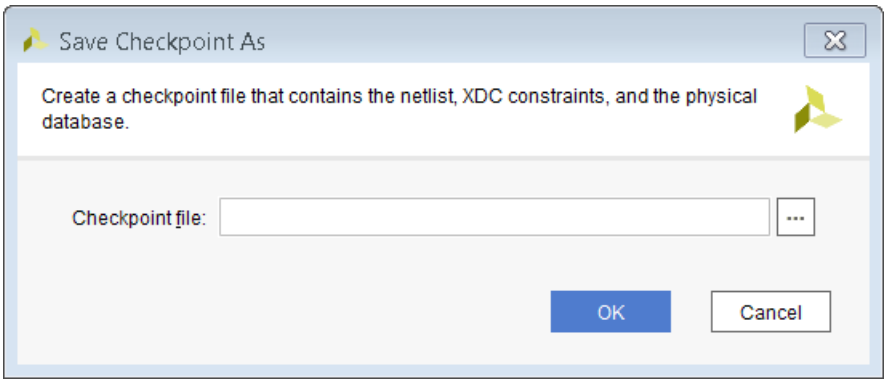
The Program section of the ECO Navigator provides access to the commands. These commands allow you to save your modifications and generate a new BIT file for programming. You can slo generate a new LTX file for your debug probes and program the device.

Figure 82: **ECO Navigator Program Commands**



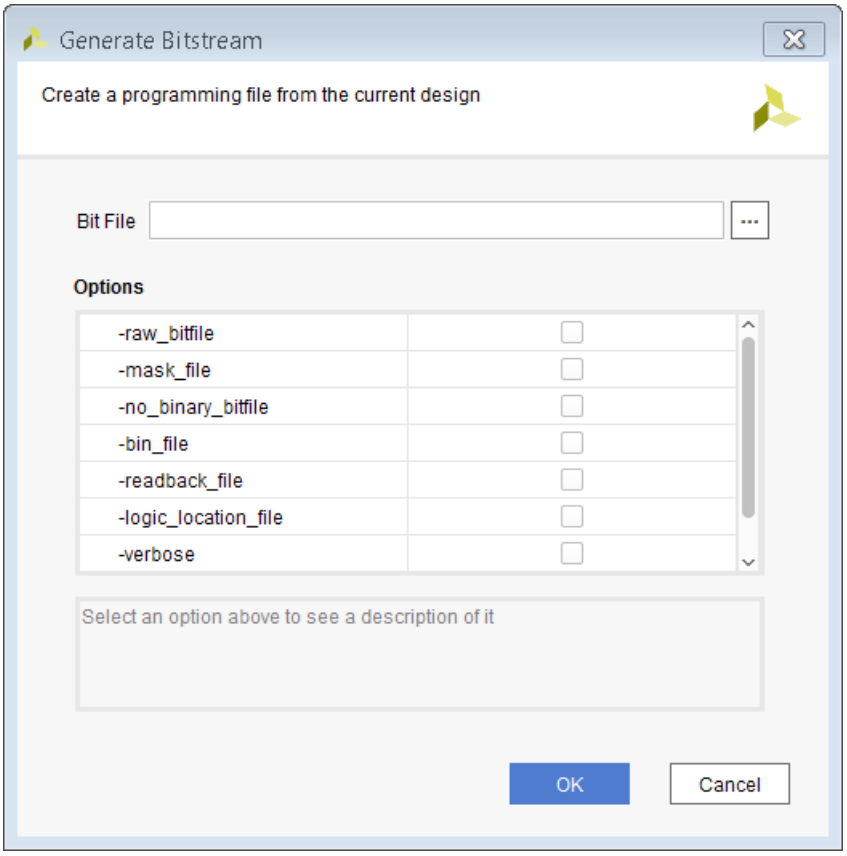
- **Save Checkpoint As:** This command allows you to save your modifications to a new checkpoint.

Figure 83: **Save Checkpoint As Dialog Box**



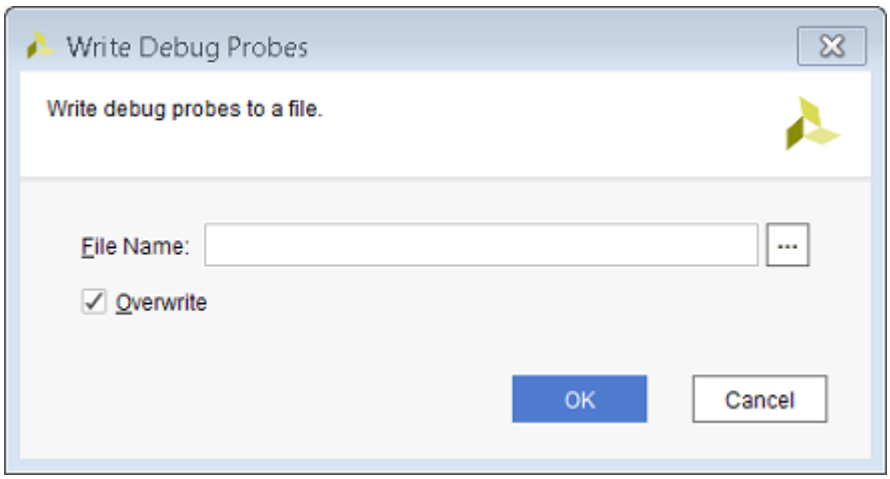
- **Generate Bitstream:** This command allows you to generate a new .bit file for programming for 7 series and UltraScale, UltraScale + devices. For Versal devices, the **Generate Device Image** command is used.

Figure 84: **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, save the updated information to a new debug probes file (LTX). It reflects the changes in the Vivado Hardware Manager.

Figure 85: **Write Debug Probes Dialog Box**



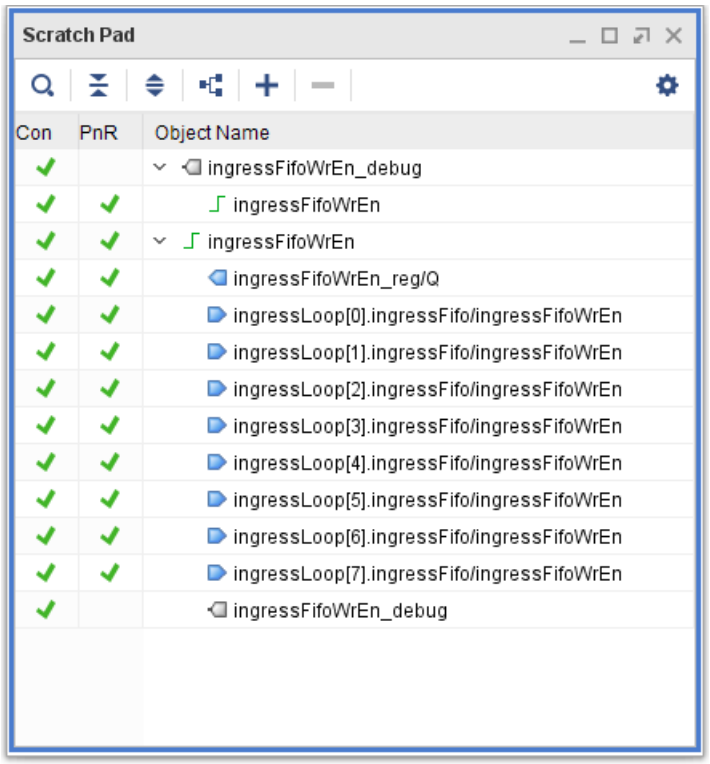
Scratch Pad

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. 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 86: **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

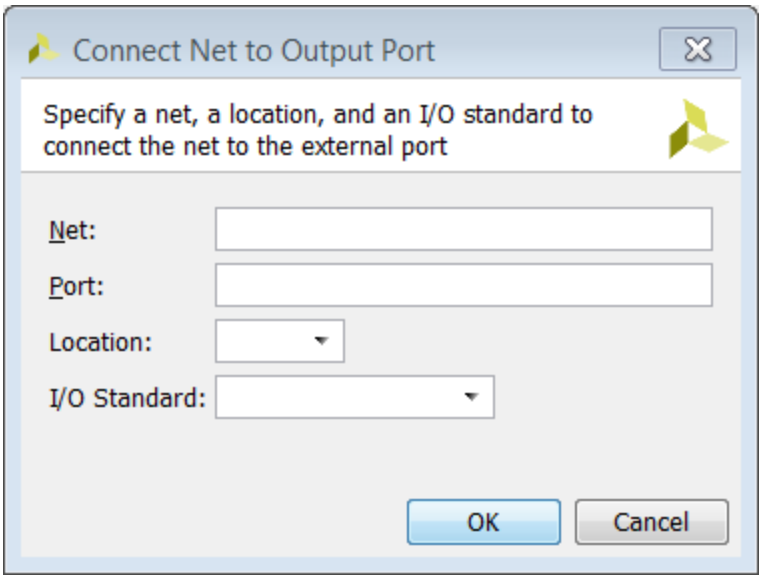
When you right-click in the Scratch Pad, the following pop-up menu commands are available:

- **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. It allows you to connect the selected net to an external port. See the following figure.
- **Elide Setting:** Specifies how to truncate long object names that do not 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 87: **Connect Net to Output Port Dialog Box**

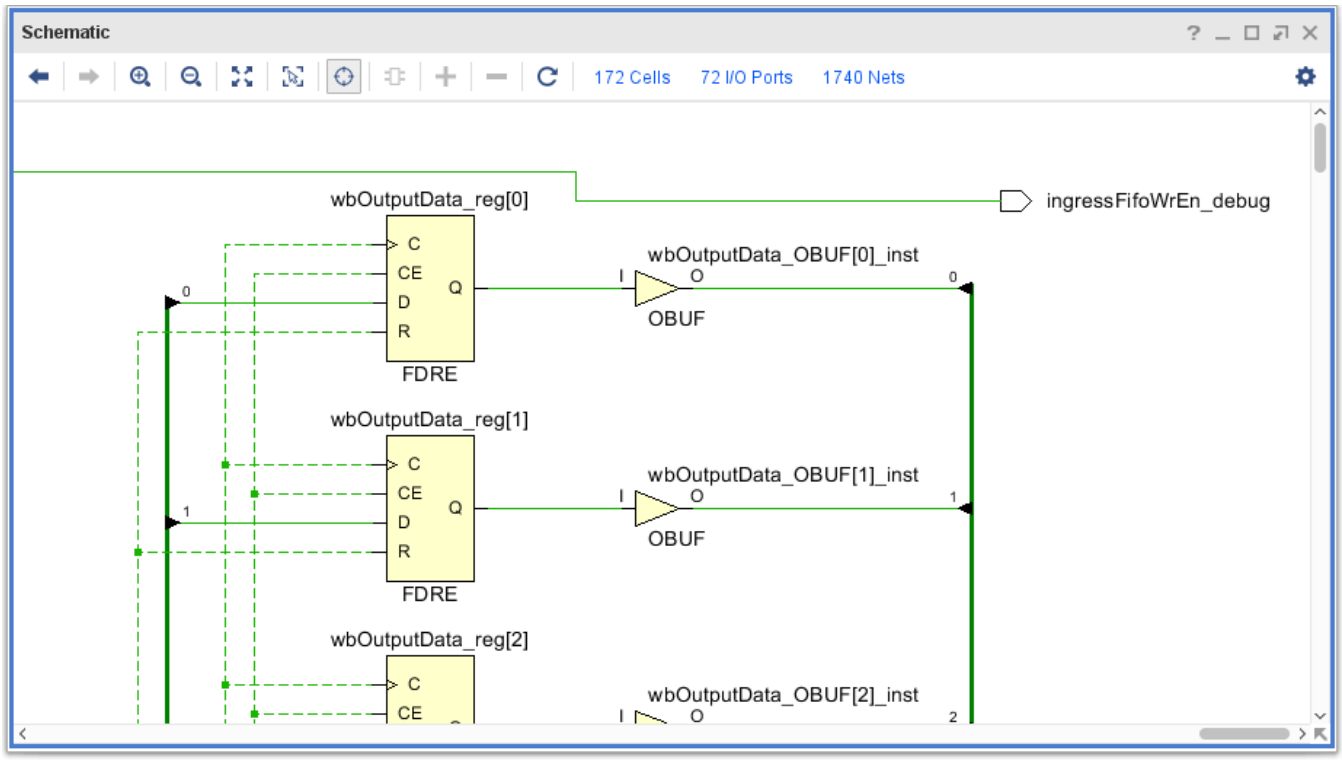


Schematic Window

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.

TIP: *Use the Mark Objects and Highlights Objects command to keep track of objects in the Schematic Window as you make changes to the netlist.*

Figure 88: **Schematic Window**



AMD

Appendix A

Using Remote Hosts and Compute Clusters

Overview

The AMD 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. It means that `$XILINX_VIVADO` and `$PATH` are configured correctly in your .cshrc/.bashrc setup scripts. The shell uses $PATH to find the Vivado executable while some XILINX tools use $XILINX_VIVADO to obtain the Vivado executable path. It is best to set both of these environment variables to the Vivado executable in your .cshrc/.bashrc setup scripts.

  Alternatively, for Manual Configuration, if Vivado is not set up at login (CSHRC or BASHRC), use the Run pre-launch script option. Specify an environment setup script to run before all jobs.

- Vivado IDE installation must be visible from the mounted file systems on remote machines. If Vivado IDE is installed on a local disk on your machine, it might not be visible to 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 the following figure. Once added, the number of jobs per host can selected and hosts can optionally be disabled. Provide the specific command used to launch the jobs.

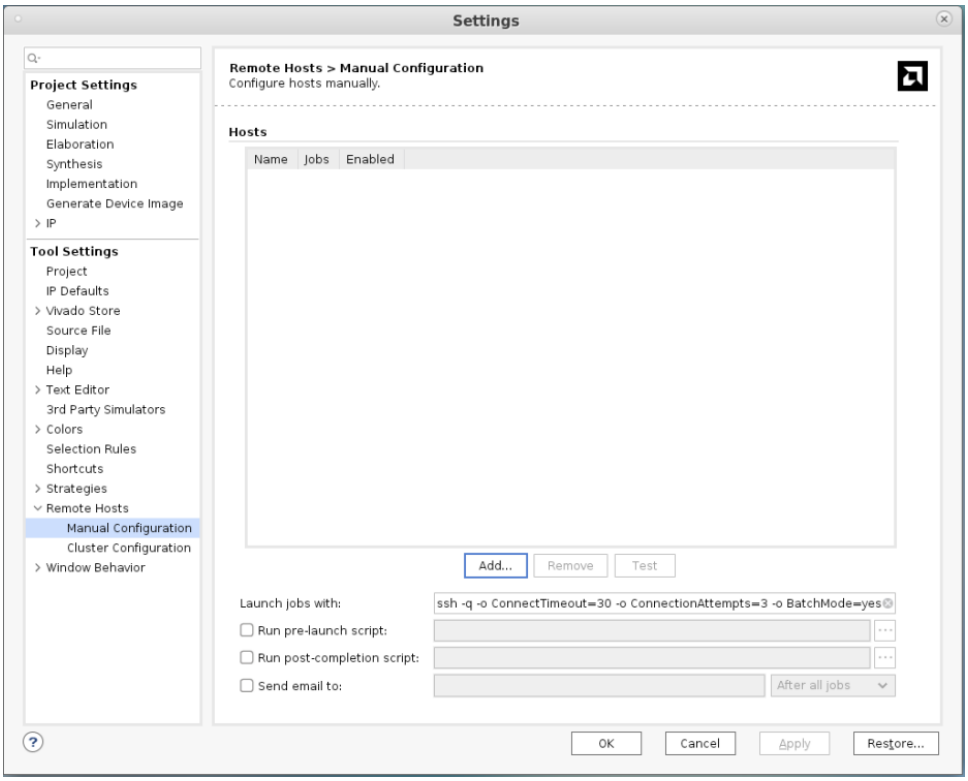
Optionally, you can configure pre- and post-launch scripts and an email address if you want the notification after the jobs complete.

IMPORTANT! *Use caution when specifying the "launch jobs with" command. For example, removing BatchMode=yes can 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 submits jobs to the remote hosts. Before launching runs on multiple Linux hosts, configure SSH so that the host does not require a password each time you launch a remote run.

Figure 89: **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: *Contact your System Administrator if you are always prompted for a password.*

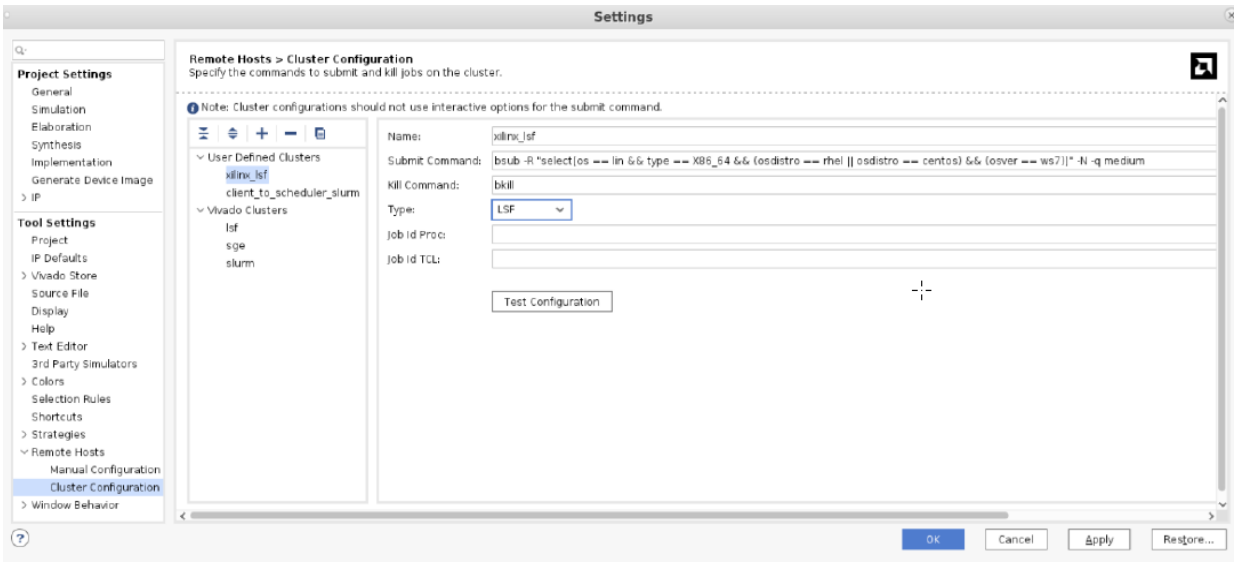
Cluster Configurations

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 supports LSF, SGE, and SLURM. For any other cluster, you can choose CUSTOM in the combo box.

For CUSTOM cluster, you must provide the path of the Tcl file, which contains logic to fetch Job ID and return Job ID value to the proc. This proc name must be used to populate the field Job ID Proc. Job ID Tcl and Job ID Proc can be left empty for natively supported clusters, that is, LSF, SGE, and SLURM. You can test the configuration by pressing the test configuration button.

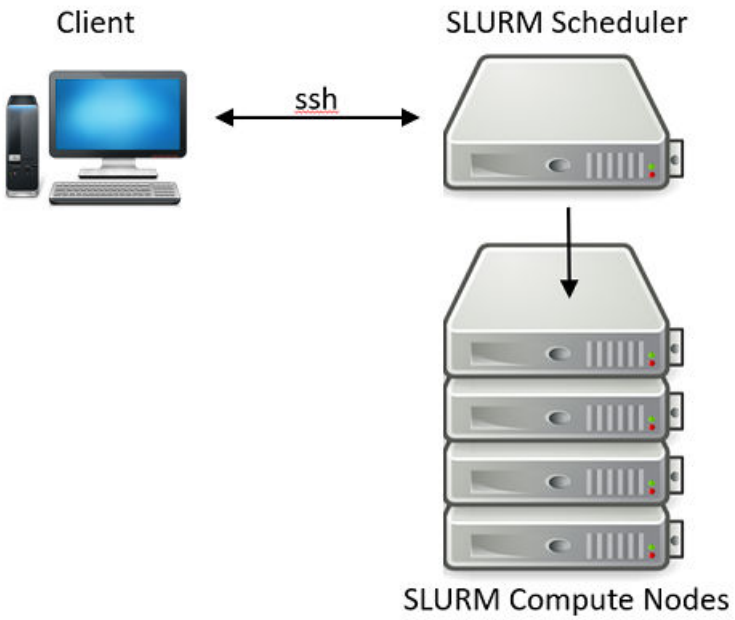
Figure 90: **Cluster Configurations Settings Dialog Box**



SLURM Specific Configuration

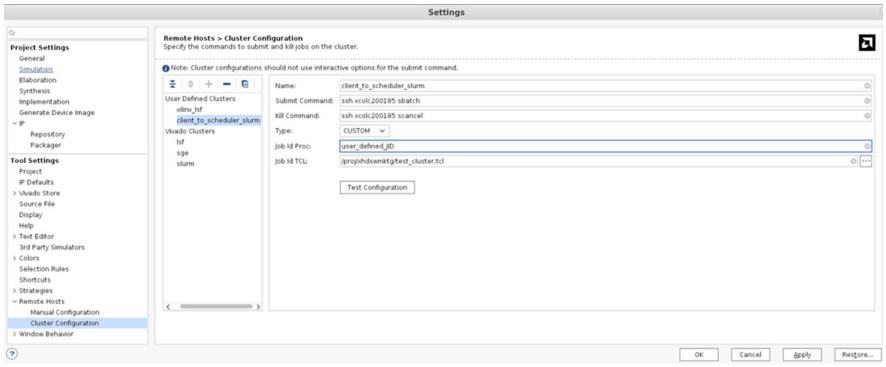
You can configure Vivado to run on SLURM using ssh to connect the client to the scheduler.

Figure 91: **SLURM Compute Nodes**



In this example, the client machine name is xcolc200189, the scheduler machine name is xcolc200185.

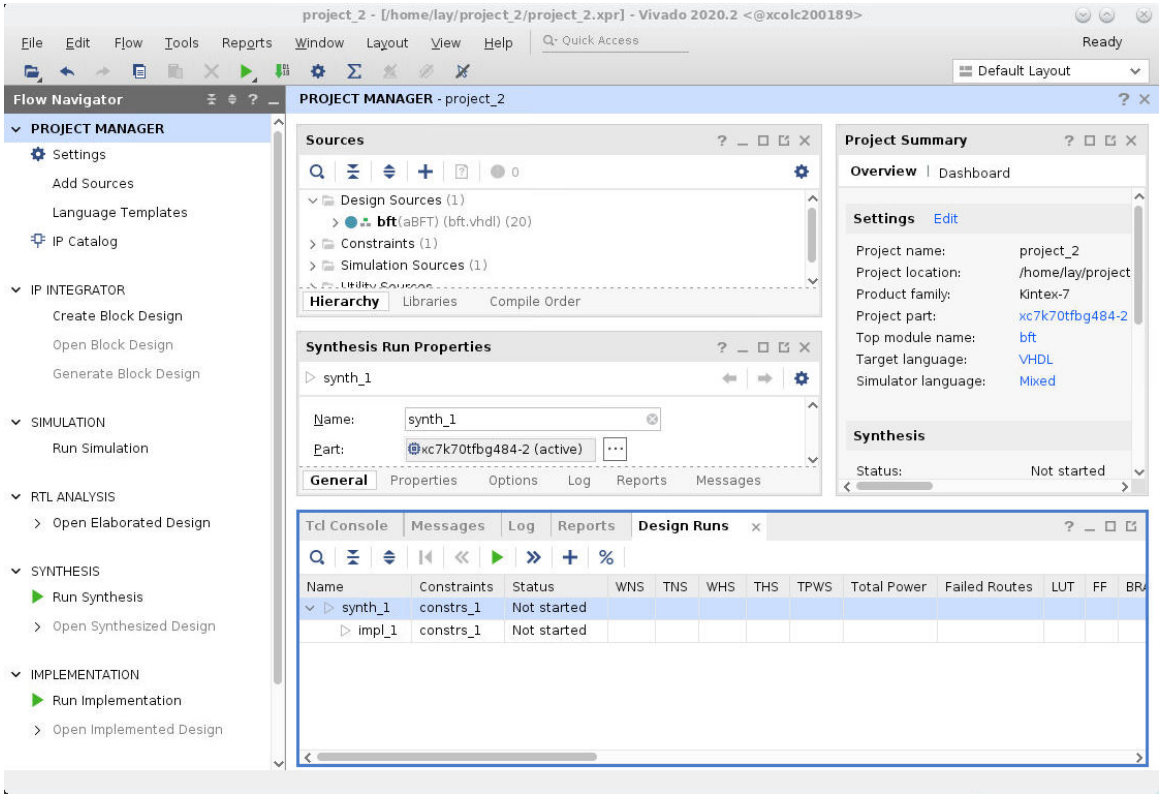
1. Set up SSH keys on client and scheduler to enable ssh without password.
2. Start Vivado on the client machine.
3. Create a custom SLURM cluster.
   a. Open the Vivado Settings dialog box (**Tools → Settings**).
   b. Select **Tool Settings → Remote Hosts → Cluster Configuration**.
   c. Click the "+" button in the toolbar to create a new cluster configuration.
   d. Fill in the form as follows. Important to leave the Type as **CUSTOM**.



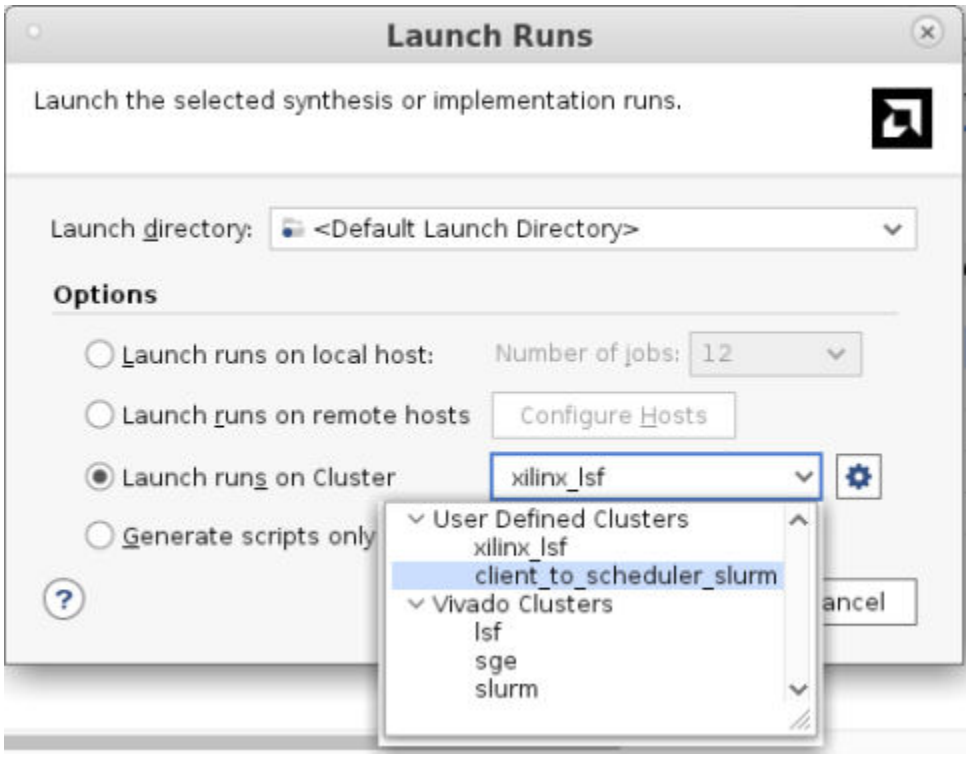
Example Tcl script to get a Job ID:

```
proc user_defined_JID {str} {
if { [regexp {(\d+)} $str matchresult] } {
return $matchresult
} else {
  return ""
 }
}
```

4. Launch a job on the cluster to test the configuration.
   a. Select **File → Project → Open Example**.
   b. Click **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.



f. In the Launch Runs dialog box, choose **Launch runs on cluster** and in the combo box, select the custom cluster name created above.

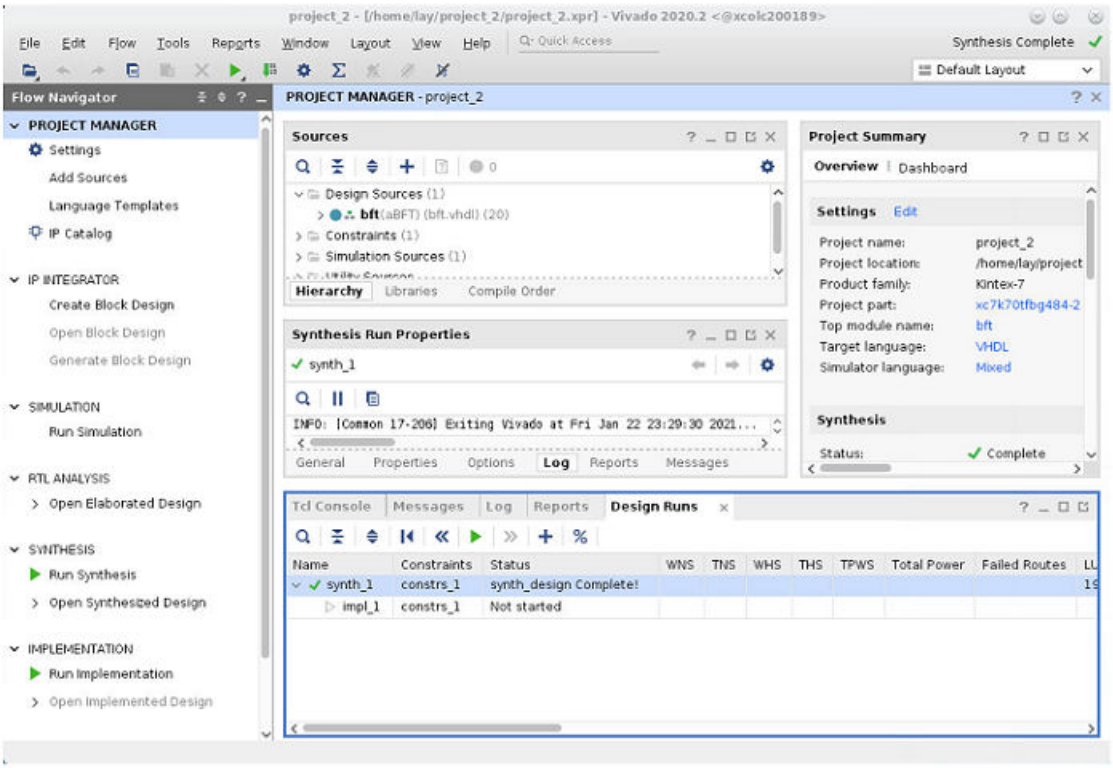


g. Click **OK** to launch the job.

h. In a terminal, ssh into the scheduler machine and check to see the job running using the `squeue` command on the scheduler machine.

```
File  Edit  View  Bookmarks  Settings  Help
[lay@xcolc200185 ~]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
                47     debug runme.sh      lay  R       0:40      1 xcolc200186
[lay@xcolc200185 ~]$
```

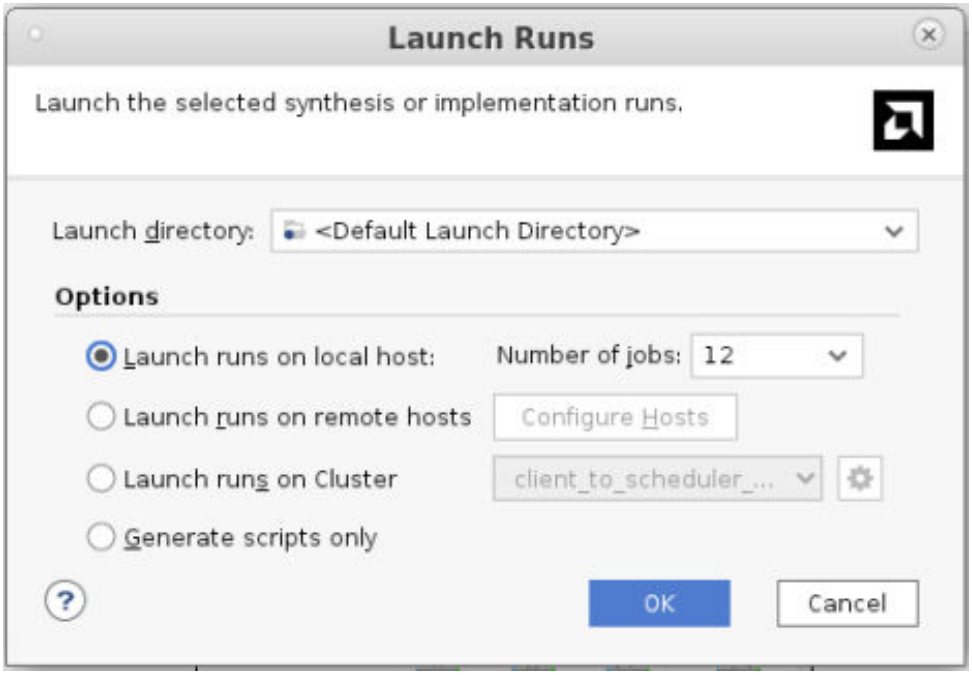
i. See the job complete successfully in the Vivado session running on the client.



Launching Jobs on Remote Hosts

Configuring remote hosts makes launching Vivado jobs easy. The following figure 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 use your preconfigured settings to execute.

Figure 92: **Launch Runs Dialog Box**



You can execute the jobs on the user configured remote hosts or clusters.

AMD

Appendix B

Implementation Categories, Strategy Descriptions, and Directive Mapping

Implementation Categories

Table 22: **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 23: **Implementation Strategy Descriptions**

Implementation Strategy Name	Description
AMD Vivado™ Implementation Defaults	Balances compile time 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_ExploreWithRemap	Similar to Performance_Explore but with logic remapped to lower logic levels.
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.

Table 23: **Implementation Strategy Descriptions** *(cont'd)*

Implementation Strategy Name	Description
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).
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.

Table 23: **Implementation Strategy Descriptions** *(cont'd)*

Implementation Strategy Name	Description
Flow_Quick	Fastest possible compile time, all timing-driven behavior disabled. Useful for utilization estimation.

Directives Used by opt_design and place_design in Implementation Strategies

Table 24: **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

Table 24: **Directives Used by opt_design and place_design in Implementation Strategies** *(cont'd)*

Strategy	opt_design -directive	place_design -directive
Flow_RuntimeOptimized	RuntimeOptimized	RuntimeOptimized
Flow_Quick	RuntimeOptimized	Quick

Directives and Switches Used by place_design in Advanced Flow

Table 25: **Directives and Switches Used by place_design in Implementation Strategies for Advanced Flow**

Strategy	place_design -directive	place_design switches and subdirectives
Performance_Explore	Explore	n/a
Performance_AggressiveExplore	AggressiveExplore	n/a
Performance_ExplorePostRoutePhysOpt	Explore	n/a
Performance_ExploreWithRemap	Explore	n/a
Performance_WLBlockPlacement	Explore	-subdirective {Floorplan.GPlace.WLDrivenBlockPlacement}
Performance_EarlyBlockPlacement	Explore	-subdirective GPlace.EarlyBlockPlacement
Performance_NetDelay_high	Explore	-net_delay_weight high
Performance_NetDelay_low	Explore	-net_delay_weight low
Performance_Retiming	Explore	n/a
Performance_ExtraTimingOpt	Explore	-subdirective { Floorplan.ExtraTimingUpdate Floorplan.ExtraTimingOpt.high GPlace.ExtraTimingUpdate GPlace.ExtraTimingOpt.high DPlace.ExtraTimingUpdate DPlace.ExtraTimingOpt.high}
Performance_BalanceSLRs	Explore	-subdirective Floorplan.BalancedSLR.high
Performance_HighUtilSLRs	Explore	-subdirective Floorplan.BalancedSLR.low
Congestion_SpreadLogic_high	Default	-subdirective { Floorplan.ForceSpreading.high GPlace.ForceSpreading.high GPlace.ReduceCongestion.high DPlace.ReducePinDensity.high}

Table 25: **Directives and Switches Used by place_design in Implementation Strategies for Advanced Flow** *(cont'd)*

Strategy	place_design -directive	place_design switches and subdirectives
Congestion_SpreadLogic_medium	Default	-subdirective { Floorplan.ForceSpreading.med GPlace.ForceSpreading.med GPlace.ReduceCongestion.med DPlace.ReducePinDensity.med}
Congestion_SpreadLogic_low	Default	-subdirective { Floorplan.ForceSpreading.low GPlace.ForceSpreading.low GPlace.ReduceCongestion.low DPlace.ReducePinDensity.low}
Congestion_SSI_Spreadlogic_high	Default	-subdirective Floorplan.BalancedSLR.high
Congestion_SSI_Spreadlogic_low	Default	-subdirective Floorplan.BalancedSLR.low
Area_Explore	Default	n/a
Area_ExploreSequential	Default	n/a
Area_ExploreWithRemap	Default	n/a
Flow_RuntimeOptimized	RuntimeOptimized	n/a
Flow_Quick	Quick	n/a

***Note*:** Performance_WLBlockPlacementFanoutOpt, RefinePlacement, SpreadSLLs, and BalanceSlls strategies are not applicable to the Advanced Flow.

Directives Used by phys_opt_design and route_design in Implementation Strategies

Table 26: **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_AggressiveExplore	AggressiveExplore	AggressiveExplore
Performance_ExplorePostRoutePhysOpt	Explore[1]	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

Table 26: **Directives Used by phys_opt_design and route_design in Implementation Strategies** *(cont'd)*

Strategy	phys_opt_design -directive	route_design -directive
Performance_Retiming	AlternateFlowWithRetiming	Explore
Performance_ExtraTimingOpt	Explore	NoTimingRelaxation
Performance_RefinePlacement	Default	Explore
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[1]	Default
Flow_RuntimeOptimized	Not enabled	RuntimeOptimized
Flow_Quick	Not enabled	Quick

Notes:

1. 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. Use Tcl to programmatically list run properties. 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; use `list_property_value` to return all supported values.

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
```

Appendix C

Additional Resources and Legal Notices

Finding Additional Documentation

Technical Information Portal

The AMD Technical Information Portal is an online tool that provides robust search and navigation for documentation using your web browser. To access the Technical Information Portal, go to https://docs.amd.com.

Documentation Navigator

Documentation Navigator (DocNav) is an installed tool that provides access to AMD Adaptive Computing documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

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

***Note*:** For more information on DocNav, refer to the *Documentation Navigator User Guide* (UG968).

Design Hubs

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

- In DocNav, click the **Design Hubs View** tab.
- Go to the Design Hubs web page.

Support Resources

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

References

These documents provide supplemental material useful with this guide:

Vivado Design Suite User Guides

1. *Vivado Design Suite User Guide: Design Flows Overview* (UG892)
2. *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893)
3. *Vivado Design Suite User Guide: Designing with IP* (UG896)
4. *Vivado Design Suite User Guide: Using Tcl Scripting* (UG894)
5. *Vivado Design Suite User Guide: System-Level Design Entry* (UG895)
6. *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994)
7. *Vivado Design Suite User Guide: Synthesis* (UG901)
8. *Vivado Design Suite User Guide: Using Constraints* (UG903)
9. *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906)
10. *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907)
11. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
12. *UltraFast Design Methodology Guide for FPGAs and SoCs* (UG949)
13. *Vivado Design Suite Properties Reference Guide* (UG912)
14. *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909)
15. *Versal Adaptive SoC Clocking Resources Architecture Manual* (AM003)
16. *Vivado Design Suite Tcl Command Reference Guide* (UG835)
17. *Versal Adaptive SoC Hardware, IP, and Platform Development Methodology Guide* (UG1387)

Other Vivado Design Suite Documents

1. *7 Series FPGAs Clocking Resources User Guide* (UG472)
2. *UltraScale Architecture Clocking Resources User Guide* (UG572)
3. *Vivado Design Suite Tcl Command Reference Guide* (UG835)

4. *ISE to Vivado Design Suite Migration Guide* (UG911)
5. *Vivado Design Suite Tutorial: Design Flows Overview* (UG888)

Vivado Design Suite Documentation Site

1. Vivado Design Suite Documentation

Training Resources

AMD 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

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
11/20/2025 Version 2025.2	
Available Logic Optimizations	Updated the section.
MBUFG Optimization	Updated the section.
place_design Command (Versal)	Added the table.
Vivado ECO Flow	Updated the section.
05/29/2025 Version 2025.1	
Available Physical Optimizations	Updated the tables.
Physical Synthesis Phase	Added Physical Optimization of SLR crossings.
Optimizing Compile Time	Added the section.
Routing	Updated the section.
Directives Used by phys_opt_design and route_design in Implementation Strategies	Updated the table.
LUT Combining in Advanced Flow (Versal) Placement	Added the section.
Run Section	Updated the section.

Please Read: Important Legal Notices

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes. THIS INFORMATION IS PROVIDED "AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

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