Lattice Radiant Software 2.1 User Guide

May 7, 2020

Copyright

Trademarks

Disclaimers

Type Conventions Used in This Document

Convention	Meaning or Use
Bold	Items in the user interface that you select or click. Text that you type into the user interface.
<*Italic*>	Variables in commands, code syntax, and path names.
Ctrl+L	Press the two keys at the same time.
`Courier`	Code examples. Messages, reports, and prompts from the software.
`...`	Omitted material in a line of code.
`.` `.` `.`	Omitted lines in code and report examples.
`[ ]`	Optional items in syntax descriptions. In bus specifications, the brackets are required.
`( )`	Grouped items in syntax descriptions.
`{ }`	Repeatable items in syntax descriptions.
`|`	A choice between items in syntax descriptions.

Contents

Chapter 1

Introduction

Lattice Radiant® software is the leading-edge software design environment for cost- sensitive, low-power Lattice Field Programmable Gate Arrays (FPGA) architectures. The Radiant software integrated tool environment provides a modern, comprehensive user interface for controlling the Lattice Semiconductor FPGA implementation process. Its combination of new and enhanced features allows users to complete designs faster, more easily, and with better results than ever before.

This user guide describes the main features, usage, and key concepts of the Radiant software design environment. It should be used in conjunction with the Release Notes and reference documentation included with the product software. The Release Notes document is also available on the Lattice Web site and provides a list of supported devices.

Radiant Software Overview

The Radiant software uses an expanded project-based design flow and integrated tool views so that design alternatives and what-if scenarios can easily be created and analyzed. The *Implementations* and *Strategies* concepts provide a convenient way for users to try alternate design structures and manage multiple tool settings.

System-level information—including process flow, hierarchy, and file lists—is available, along with integrated HDL code checking and consolidated reporting features.

A fast Timing Analysis loop and Programmer provide capabilities in the integrated framework. The cross-probing feature and the shared memory architecture ensure fast performance and better memory utilization.

The Radiant software is highly customizable and provides Tcl scripting capabilities from either its built-in console or from an external shell.

The Radiant software has many of the same features as Lattice Diamond software, and adds new features, such as:

- Constraints support utilizing industry standard SDC format.
- Efficient, easy-to-use integrated graphical user interface (GUI) with a new look-and-feel that gives users more efficient access to popular tools.

- Unified timing analysis engine with enhanced timing reports for faster design timing closure.

User Guide Organization

This user guide contains all the basic information for using the Radiant software. It is organized in a logical sequence from introductory material, through operational descriptions, to advanced topics.

Key concepts and work flows are explained in “Design Environment Fundamentals” on page 17 and “Radiant Software Design Flow” on page 56.

Basic operation of the design environment is described in “User Interface Operation” on page 23.

The chapter “Working with Projects” on page 42 shows how to set up project implementations and strategies.

The chapter “Working with Tools and Views” on page 71 describes the many tool views available.

Chapter 2

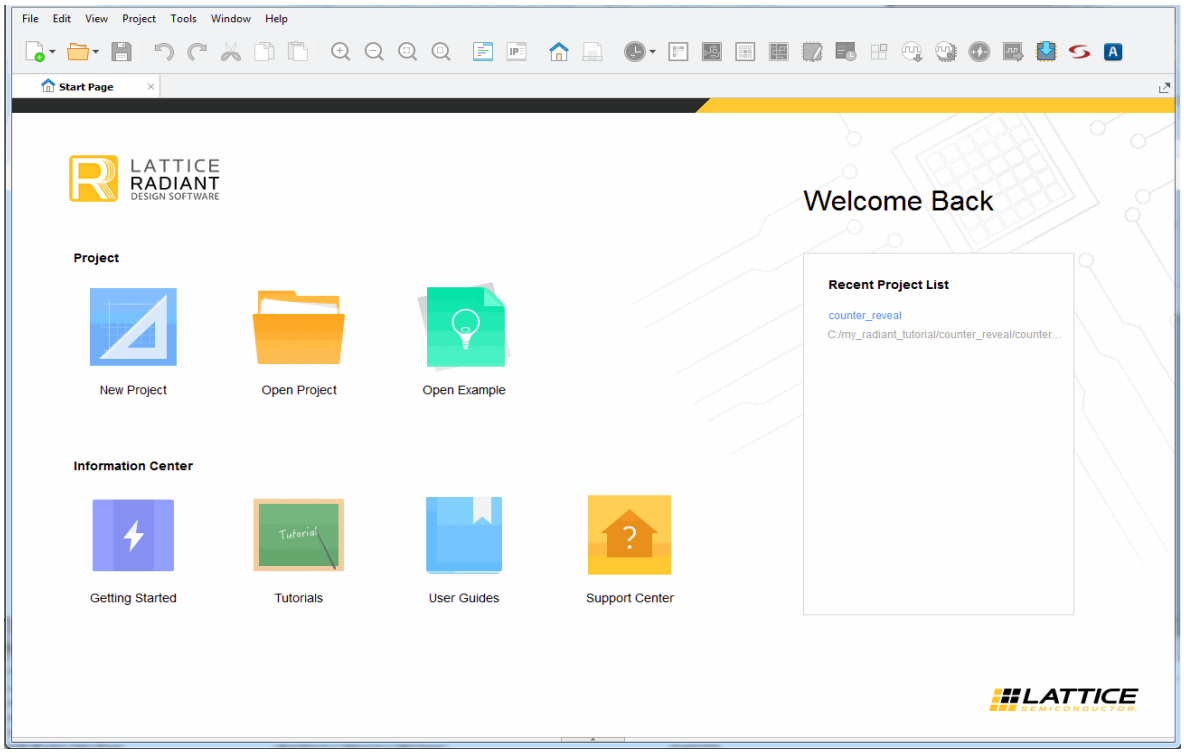
Getting Started

This chapter explains how to run the Radiant software and open or create a project. For more information about project fundamentals, see the chapters "Design Environment Fundamentals" on page 17 and "Working with Projects" on page 42.

Prerequisites

To run the Radiant software, select **Radiant Software** from the installation location. This opens the default Start Page, shown in Figure 1.

Figure 1: Default Start Page



Creating a New Project

A project is a collection of all files necessary to create and download your design to the selected device. The New Project wizard guides you through the steps of specifying a project name and location, selecting a target device, and adding existing sources to the new project.

Note

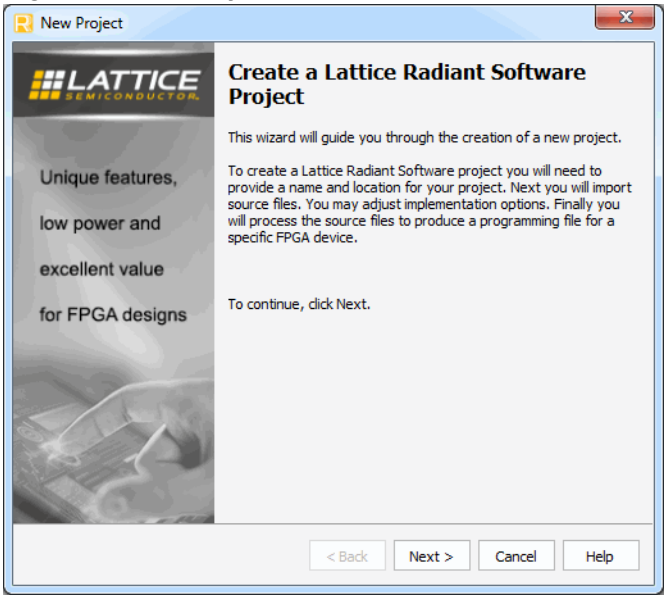
Do not place more than one project in the same directory.

To create a new project:

1. From the Radiant main window, click the **New Project** button, or choose **File > New > Project**.

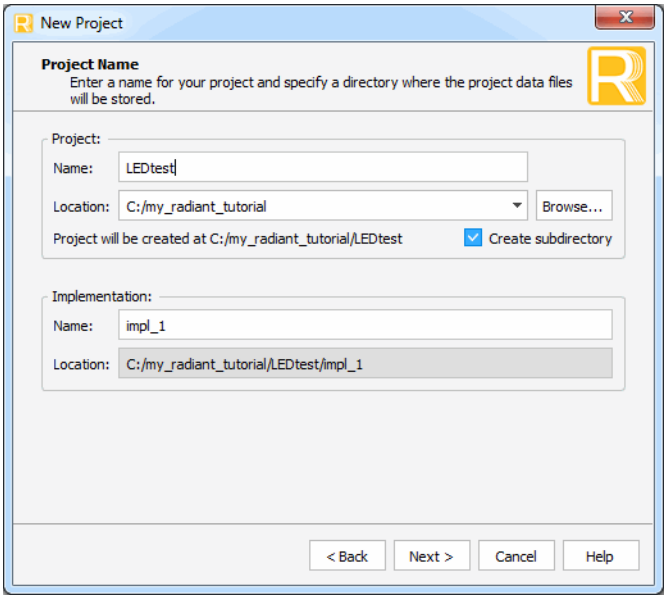
   The New Project confirmation window opens, shown in Figure 2.

Figure 2: New Project Confirmation Window



2. Click **Next**. The New Project wizard opens, shown in Figure 3.

Figure 3: New Project Wizard



3. In the Project Name dialog box, do the following:
   - Under Project, specify the name for the new project.

     File names for Radiant software projects and project source files must start with a letter (A-Z, a-z) and must contain only alphanumeric characters (A-Z, a-z, 0-9) and underscores (_). Spaces are allowed.
   - To specify a location for your project, click **Browse**. In the Project Location dialog box, you can specify a desired location.
   - Under Implementation, specify the name for the first version of the project. You can have more than one version, or "implementation," of the project to experiment with. For more information on implementations, refer to "Implementations" on page 44.
   - To create a sub-directory with the same name as your location directory, click **Create Subdirectory**. This will allow you to keep your project implementations separate. If this box is left unchecked, no sub-directory will be created in the project directory.
   - When you finish, click **Next**.
4. In the Add Source dialog box, do the following if you have an existing source file that you want to add to the project. If there are no existing source files, click **Next**.
   a. Click **Add Source**. You can import HDL files at this time.In the Import File dialog box, browse for the source file you want to add, select it, and click **Open**.

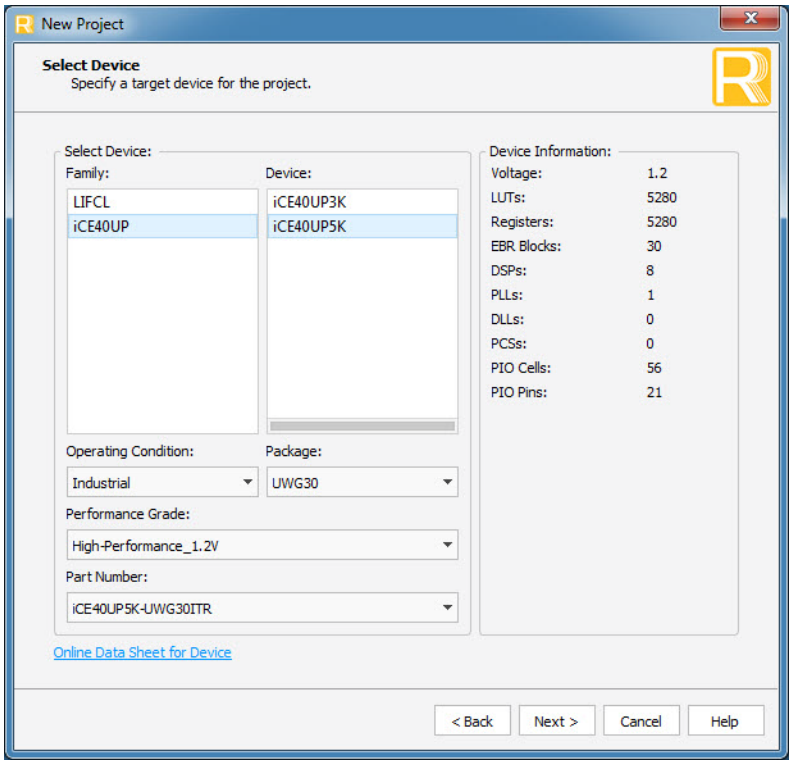
      The source file is then displayed in the Source files field.
   b. Repeat the above to add more files.
   c. To copy the added source files to the implementation directory, select **Copy source to implementation directory**. If you prefer to reference these files, clear this option.

d. To create empty Lattice Design Constraint (.ldc) file and Physical Constraint File (.pdc) files that can be edited at a later time, select **Create empty constraint files.** Refer to the chapter "Implementations" on page 44 for more information about constraint files.

e. When you finish, click **Next**.

5. In the Select Device dialog box, shown in Figure 4, select a device family and a specific device within that family. Then choose the options you want for that device. When you finish, click **Next**.

Figure 4: Select Device Dialog Box



6. In the Select Synthesis Tool dialog box, select the synthesis tool that you want to use. This choice can be changed at any time. When you finish, click **Next**.

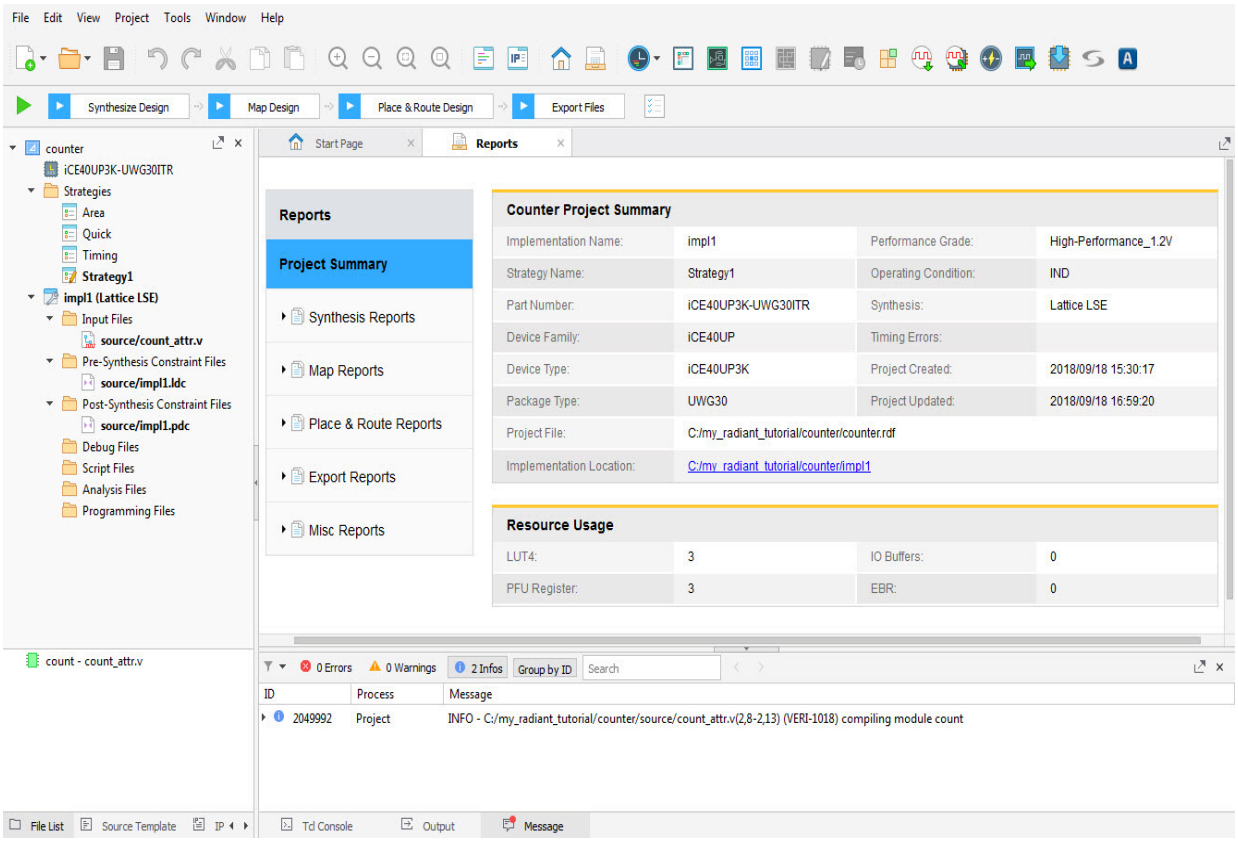
7. In the Project Information dialog box, make sure the project settings are correct.

Note

If you want to change some of the settings, click **Back** to modify them in the previous dialog boxes of the New Project Wizard.

Click **Finish**. The newly created project, shown in Figure 5, is now created and open.

Figure 5: Opened Project



Select the **File List** tab under the left pane, to view the Test project file list.

To close a project, choose **File > Close Project**.

Opening an Existing Project

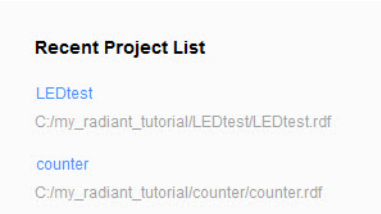
Use one of the following methods to open an existing Radiant software project:

- On the Start Page, click the **Open Project** button.
- From the File menu, choose **Open > Project**.
- On the Start Page, select the desired project from the Recent Projects List. Alternatively, choose a recent project from the **File > Recent Projects** menu.

You can use the Options dialog box to increase the number of projects that are shown in the Recent Projects list and to automatically load the previous project at startup. Choose **Tools > Options** to open the dialog box. To increase the number of recent projects listed, click the **General** tab and enter a number for “Maximum items shown in Recent Project List” (up to 32). To automatically open the previous project during startup, click the **Startup** tab

and then choose **Open Previous Project** from the “At Lattice Radiant Software startup” menu.

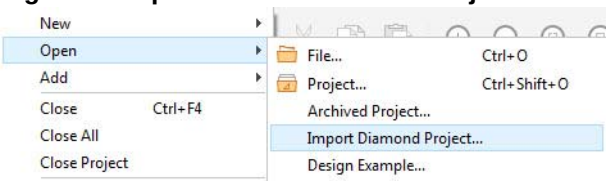
Figure 6: Recent Project List



Importing a Lattice Diamond Project

To import a Lattice Diamond project into the Radiant software, choose **File > Open > Import Diamond Project**.

Figure 7: Import Lattice Diamond Project



The file browser applies an ***.ldf** file filter to help you find Lattice Diamond project files. The Lattice Diamond project is converted to a Radiant software project.

For more information about importing Lattice Diamond projects into the Radiant software, refer to the *Lattice Radiant Software Guide for Diamond Users*.

Next Steps

After you have a project opened in the Radiant software, you can go sequentially through the rest of this user guide to learn how to work with the entire design environment, or you can go directly to any topic of interest.

- The chapters “Design Environment Fundamentals” on page 17 and “Radiant Software Design Flow” on page 56 provide explanations of key concepts.
- “User Interface Operation” on page 23 provides descriptions of the functions and controls that are available in the Radiant software environment.
- The chapters “Working with Projects” on page 42 and “Working with Tools and Views” on page 71 explain how to run processes and use the design tools.

- "Tcl Command Reference Guide" on page 147 provides an introduction to the scripting capabilities available, plus command-line shell examples.
- "Advanced Topics" on page 183 provides further details about environment options, shared memory, and Tcl scripting.

Chapter 3

Design Environment Fundamentals

This chapter provides background and discussion on the technology and methodology underlying the Radiant software design environment. Important key concepts and terminology are defined.

Overview

Understanding some of the fundamental concepts behind the Radiant software framework technology will increase your proficiency with the tool and allow you to quickly come up to speed on its use.

The Radiant software is a next-generation software design environment that uses a new project-based methodology. A single project can contain multiple implementations and strategies to provide easily managed alternate design structures and tool settings.

The process flow is managed at a system level with run management controls and reporting. Context-sensitive views ensure that you only see the data that is available for the current state in the process flow.

The shared memory technology enables many of the advanced functions in the Radiant software. Easy cross-probing between tool views and faster process loops are among the benefits.

Note

By loading the Radiant software multiple times, you can run different Radiant projects simultaneously. However, you must not load the same project in more than one Radiant software instance, as software conflicts can occur.

The Radiant software can also be run remotely. Refer to the *Lattice Radiant Software Installation Guide for Windows* or *Lattice Radiant Software Installation Guide for Linux* for more information.

Project-Based Environment

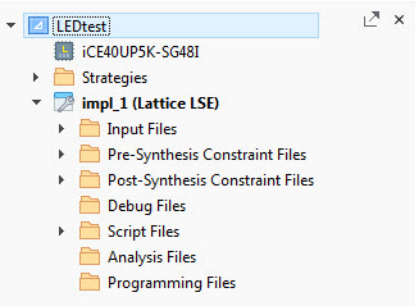
A project in the Radiant software consists of the following file types:

- HDL source files
- Constraint files
- Reveal debug files
- Script files for simulation
- Analysis files for power calculation and timing analysis
- Programming files

The Radiant software also includes settings for the targeted device and the different tools. The project data is organized into implementations, which define the project structural elements, and strategies, which are collections of tool settings.

The following File List shows the items in a sample project.

Figure 8: File List



- LEDtest: Project Name
- iCE40UP5K-SG48I: Device Part Number
- Strategies: Strategy Types
- impl_1 (Lattice LSE): Implementation (Synthesis Tool)
- Input Files: Input File Library
- Pre-Synthesis Constraint Files: Pre-Synthesis Constraint File Library
- Post-Synthesis Constraint Files: Post-Synthesis Constraint File Library
- Debug Files: Used for Reveal debugging
- Script Files: Used for Simulation Wizard
- Analysis Files: Used for Power Calculator and Timing Wizard
- Programming Files: Used for programming a device

Each item that is displayed in **bold** means that it has been selected as the active item for an implementation. An implementation displayed in **bold** means that it has been selected as the currently active implementation for the project. Your project must have one active implementation, and the implementation must have one active strategy. Optional items, such as Reveal hardware debugger files, can be set as active or inactive.

The project is the top-level organizational element in the Radiant software, and it can contain multiple implementations and multiple strategies. This enables you to try different design approaches within the same project. If you want to have a Verilog version of your design, for example, make an implementation that consists of only the Verilog source files. If you want another version of the design with primarily Verilog files but a Structural Verilog (.vm) netlist for one module, create a new implementation using the Verilog and .vm source files.Each implementation can have Verilog, VHDL or Structural Verilog source or mixed of them. The same project and design is used, but with a different set of modular blocks.

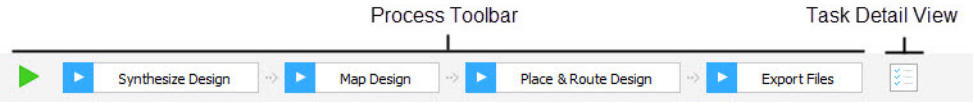
Similarly, if you want to try different implementation tool options, you can create a new strategy with the new option values.

You manage these multiple implementations and strategies for your project by setting them as active. There can only be a single active implementation with its one active strategy at a time.

Process Flow

A process is a specific task in the overall processing of a source or project. Typical processing tasks include synthesizing, mapping, placing, and routing. You can view the available processes for a design in the Process Toolbar.

Figure 9: Process Toolbar



Click the Task Detail View to see detailed information of the processes.

Processes are grouped into categories according to their functions.

- **Synthesize Design**

  Click on this process and Lattice Synthesis Engine (LSE) runs to synthesize the design. By default, this process runs the LSE tool.

  If you are using Synplify Pro, choose Synplify Pro as the synthesis tool (**Project > Active Implementation > Select Synthesis Tool**).

  - **Post-Synthesis Timing Analysis**

    Runs timing analysis after the Synthesize Design process.

  - **Post-Synthesis Simulation File**

    Generates a netlist file <*file_name*>_syn.vo used for functional verification.

- **Map Design**

  This process maps a design to an FPGA. Map Design is the process of converting a design represented as a network of device-independent components (such as gates and flip-flops) into a network of device-specific components (for example, configurable logic blocks).

  - **Map Timing Analysis**

    Runs timing analysis after the Map Design process.

- **Place & Route Design**

  After a design has undergone the necessary translation to bring it into the Unified Database (.udb) format, you can run the Place & Route Design process. This process takes a mapped physical design .udb file, places and routes the design, and then outputs a file that can then be processed by the design implementation tools.

- **Place & Route Timing Analysis**

  Runs timing analysis after Place & Route process.

- **I/O Timing Analysis**

  Runs I/O timing analysis that allows you to view the path delay tables and Timing Analyzer report of your timing constraints after placement and routing.

- **Export Files**

  You can check the desired file you want to export and run this process.

  - **Bitstream File**

    This process takes a fully routed physical design as input and produces a configuration bitstream (bit images). The bitstream file contains all of the configuration information from the physical design defining the internal logic and interconnections of the FPGA, as well as device-specific information from other files associated with the target device.

  - **IBIS Model**

    This process generates a design-specific IBIS (I/O Buffer Information Specification) model file (<*project_name*>.ibs).

    IBIS models provide a standardized way of representing the electrical characteristics of a digital IC's pins (input, output, and I/O buffers).

  - **Gate-Level Simulation File**

    This process backannotates the routed design with timing information so that you may run a simulation of your design. The backannotated design is a Verilog netlist.

  The Reports view allows you to examine and print process reports.

  Messages are displayed in the Messages window at the bottom of the Radiant software main window.

The process status icons are defined as follows:

 Process in initial state (not processed)

 Process completed successfully

 Process completed with unprocessed subtasks

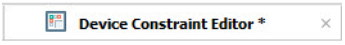
 Process failed

Shared Memory

The Radiant software uses a shared memory architecture. All tool and data views look at the same design data at any point in time. This means that when you change a data element in one view of your design, all other views will see the change, whether they are active or not.

When project data has been changed but not yet saved, an asterisk (*) is displayed in the title tab of the view.

Figure 10: Title Tab with Changed Content Indication



Notice that the asterisks indicating changed data will appear in all views referencing the changed data.

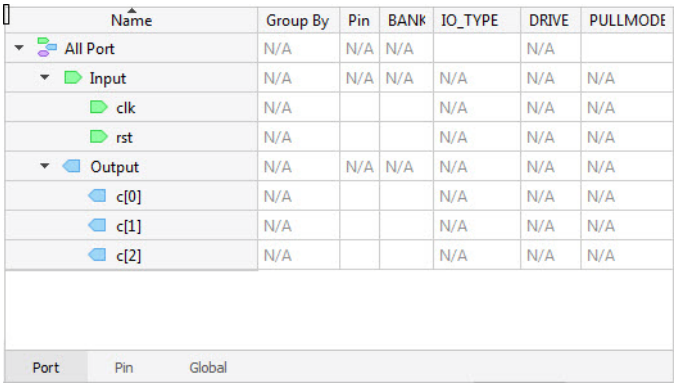
If a tool view becomes unavailable, the Radiant software environment will need to be closed and restarted.

Context-Sensitive Data Views

The data in shared memory reflects the state or context of the overall process flow. This means that views such as Device Constraint Editor Spreadsheet View will display only the data that is currently available, depending on process steps that have been completed.

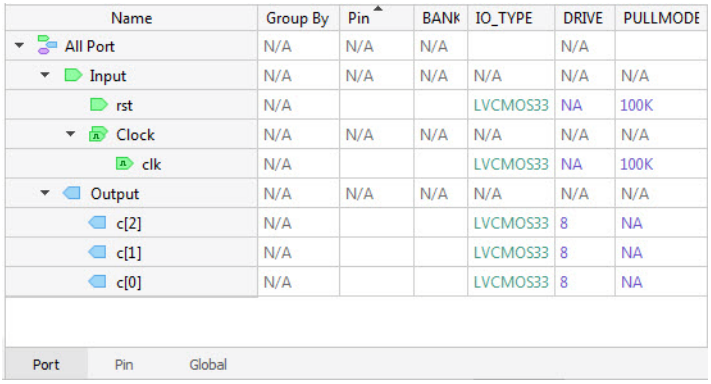
For example, Figure 11 shows the Process flow before Synthesis. Therefore, Spreadsheet View shows no IO Type or PULLMODE.

Figure 11: Process Completed Before Synthesis



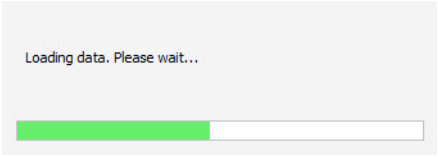
After Synthesis has been completed, Spreadsheet View displays IO Type and PULLMODE assignments, as shown in Figure 12.

Figure 12: Process Completed Through Synthesis



When you see the “Loading Data” message displayed in Figure 13, it means that a process has been completed and that the shared memory is being updated with new data.

Figure 13: Loading Data



All tool views are dynamically updated when new data becomes available. This means that when you rerun an earlier process while a view is open and displaying data, the view will remain open but dimmed because its data is no longer available.

Chapter 4

User Interface Operation

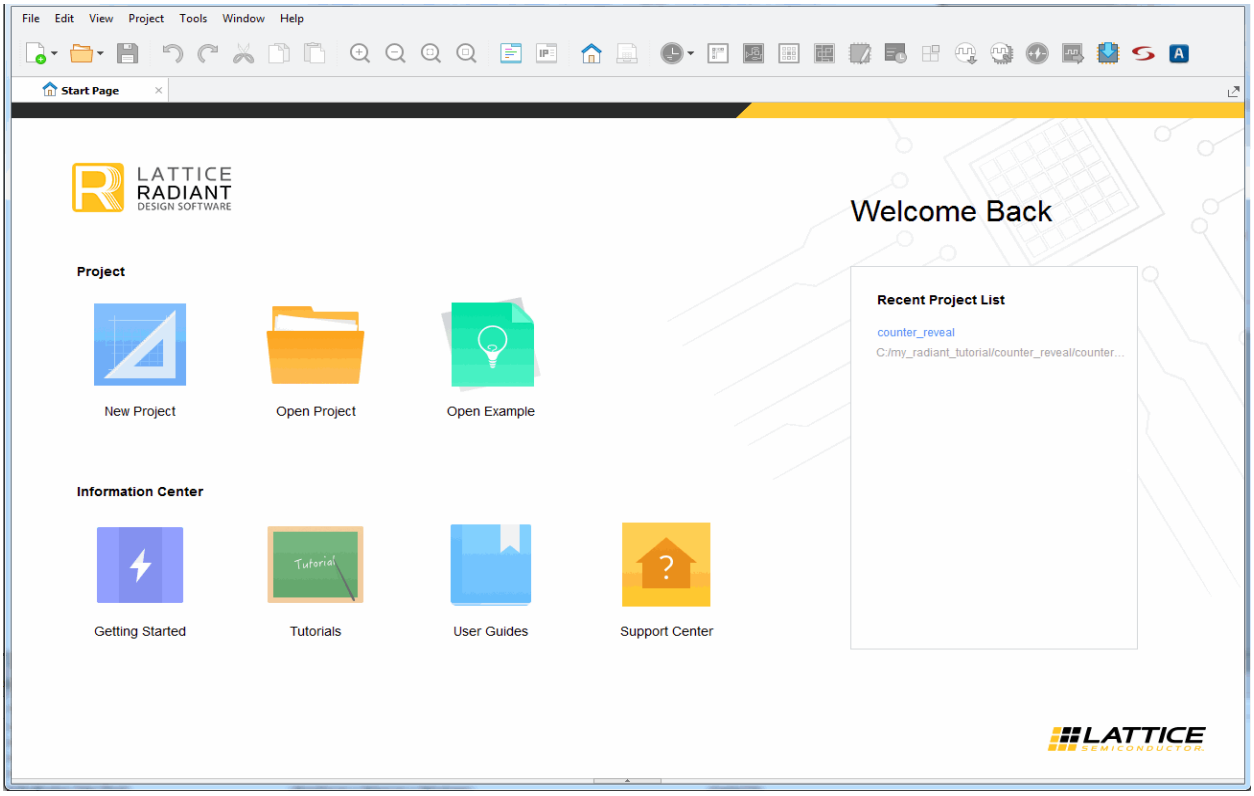
This chapter describes the user interface features, controls, and basic operation of the Radiant software. Each major element of the interface is explained. The last section in the chapter describes common user interface tasks.

Overview

The Radiant software user interface (UI) provides a comprehensive, integrated tool environment. The UI is very flexible and configurable, enabling you to store constraints for the layout you choose.

This chapter will take you through the operation of the main elements of the UI, but you should also explore the controls at your own pace. Figure 14 shows the Radiant software main window in the default state.

Figure 14: Default Start Page



Start Page

The Start Page contains three major sections, as shown in Figure 14.

- **Project**: This section allows you to create a new project; open an existing Project, and open an example.
- **Information Center:** This section has a links to Getting Started, Tutorials, User Guides, and Support Center.
- **Recent Project List:** Provides a quick way to load a recent project you've been working on.

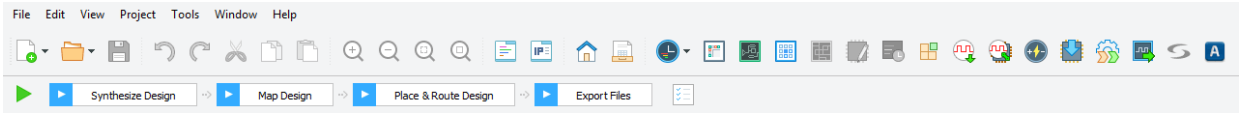
The Start Page appears in the View area by default when the Radiant software is first launched, and can be opened from the **View** tab on the menu.

The Start Page can be closed, opened, detached, and attached using the Attach button. See "Basic UI Controls" on page 28.

Menus and Toolbars

At the top of the main window is the menu and toolbar area. High-level controls for accessing tools, managing files and projects, and controlling the layout are contained here. All toolbar functionality is also contained in the menus. The menus also have functions for system, project and toolbar control.

Figure 15: Menu and Toolbar Area



The toolbars are organized into functional sets. The display of each toolbar is controlled in the **View > Toolbar** menu and also by right-clicking in the Menu and Toolbar area. The Process Toolbar lists all the processes available, such as Synthesize Design, Map Design, Place & Route Design, and Export Files. A process is a specific task in the overall processing of a source or project. You can view the available processes for a design in the Process Toolbar.

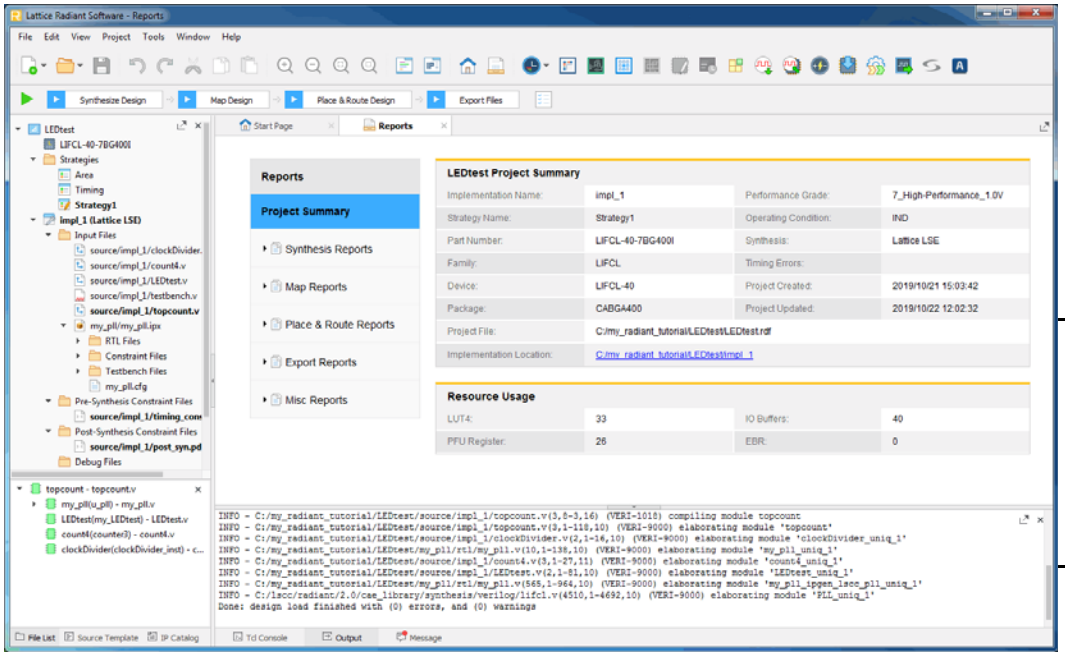
Click **Task Detail View** to see detailed information of the processes available.

Reports and Messages Views

The Reports view allows you to examine and print process reports. There are two panes in the Reports view. The left pane lists the reports. The right pane displays the reports.

Log messages are displayed in the Output frame of the Radiant software main window.

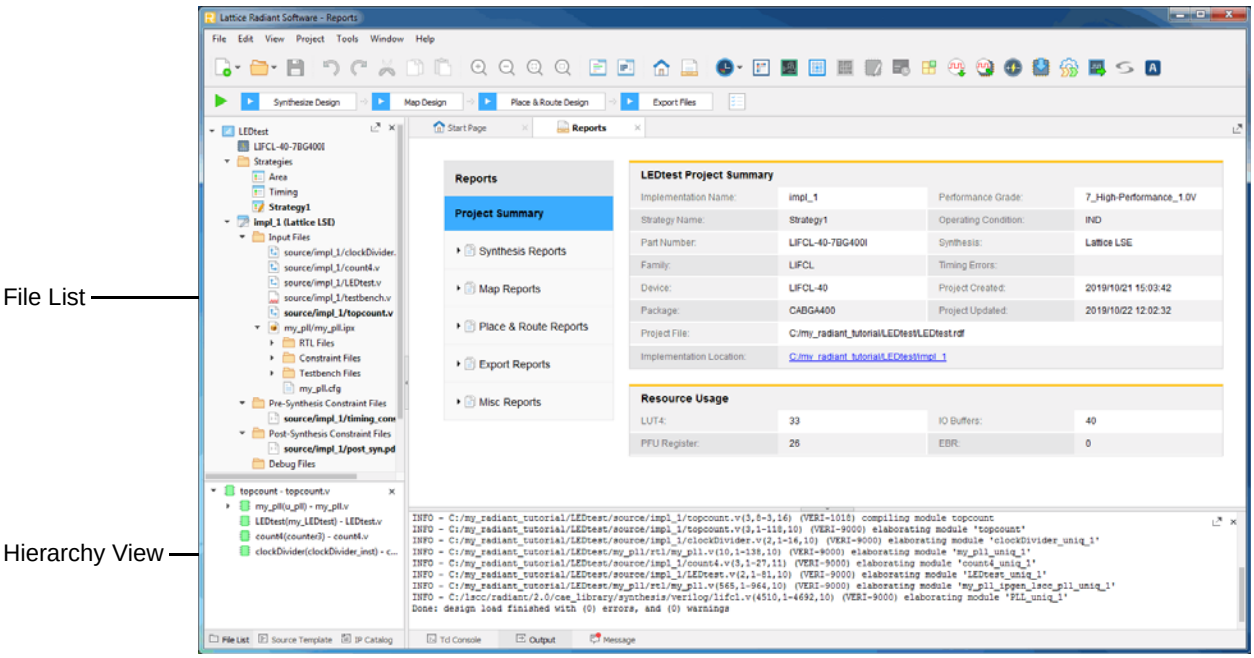
Figure 16: Reports and Log Message Views



File List Views

In the middle of the main window on the left side is the File List area. This is where the overall project and process flow is displayed and controlled.

Figure 17: File List Area



Tabs at the bottom of the File List area allow you to select between the following views:

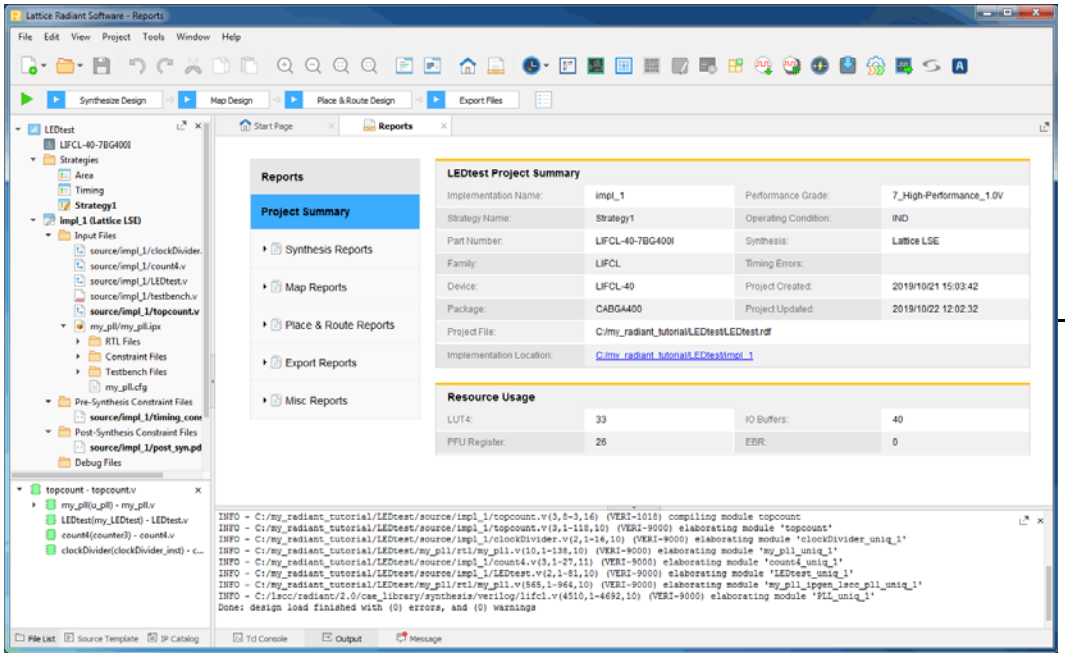
- File List – shows the files in the project organized by implementations and strategies. This is not a hierarchical listing of the design.
- Source Template – provides templates for creating VHDL, Verilog, and Constraint files.
- IP Catalog – lists available modules/intellectual properties (IP).

Underneath the File List is the Hierarchy View area. It allows you to view the hierarchical design representation. Hierarchy view shares the left pane with File List view.

Tool View Area

In the middle of the main window on the right side is the Tool View area. This is where the Start Page, Reports View, and all the Tool views are displayed.

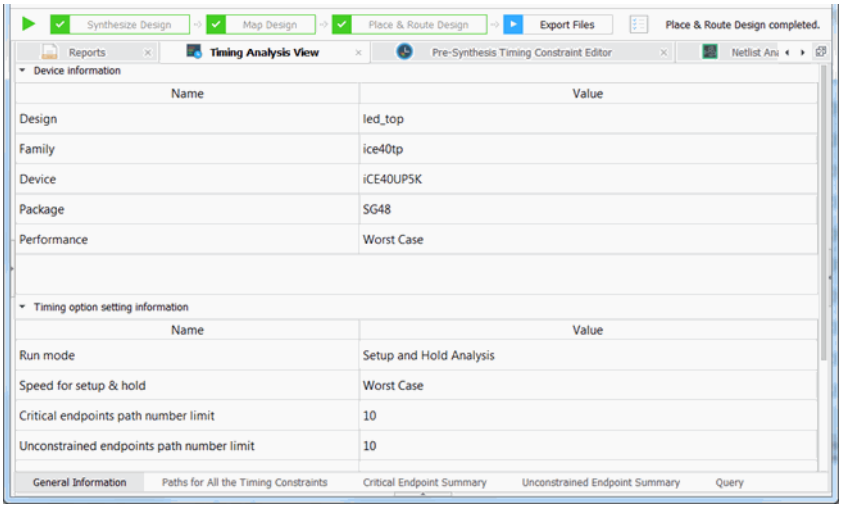
Figure 18: Tool View Area



Multiple tools can be displayed at the same time. The tool tabs include controls for grouping the tool views as well as integrating all tool views back into the main window.

Each tool view is specific to its tool and can contain additional toolbars and multiple panes or windows controlled by additional tabs. The chapter "Working with Tools and Views" on page 71 provides more details about each tool and view.

Figure 19: Multiple Tools

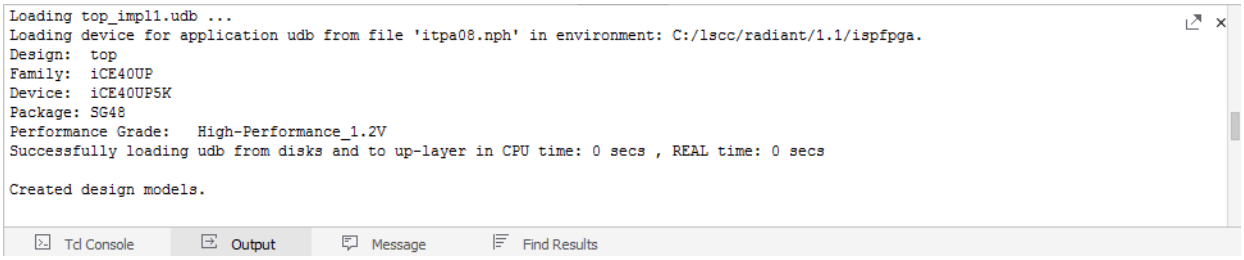


Output and Tcl Console

Near the right bottom of the main window is the Tcl Console, Output, and Message area.

Tabs at the bottom of this area allow you to select between Tcl Console, Output, and Message. Tool output is automatically displayed in the Output tab, and Errors and Warnings in the Message tab.

Figure 20: Output and Tcl Console Area



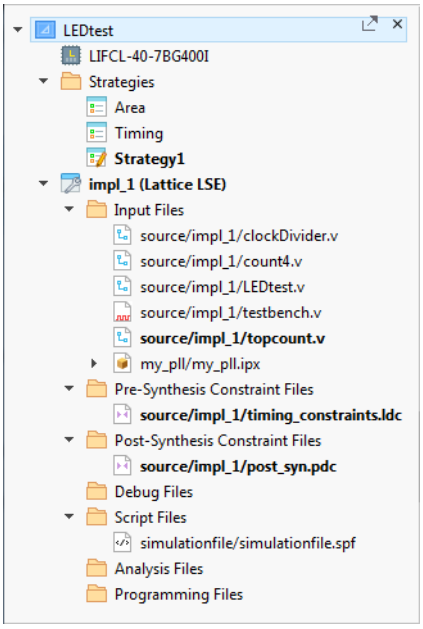
Basic UI Controls

The Radiant software environment is based on modern industry standard user interface concepts. The menus, toolbars, and mouse pointer all behave in familiar ways. You can resize any of the window panes, drag and drop elements, right-click a design element to see available actions, and hold the mouse pointer over an object to view the tool tip.

File List

The File List is a project view that shows the files in the project, including implementations and strategies. It is not a hierarchical listing of the design, but rather a list of all the design source, configuration and control files that make up the project.

Figure 21: File List



At the top of the File List is the project name. Directly below the project name is the target device, followed by the strategies, and then the implementations. There must be one active implementation, and it must have one active strategy. Active elements are indicated in **bold**.

You can right-click any file or item in the File List to access a pop-up menu of currently available actions for that item. The pop-up menu contents vary, depending on the type of item selected.

The File List view can be hidden by clicking the small arrow in right border: “Click to show/hide side panel.”

Source Template

The Source Template is a project view that provides templates for creating VHDL, Verilog, and constraint files. Templates increase the speed and accuracy of design entry. You can drag and drop a template directly to the source file. You can also create your own templates.

To access templates, choose **View > Show Views > Source Template**, or click icon in the toolbar, or click on the **Source Template** tab in the bottom-left pane, to locate and access the following templates:

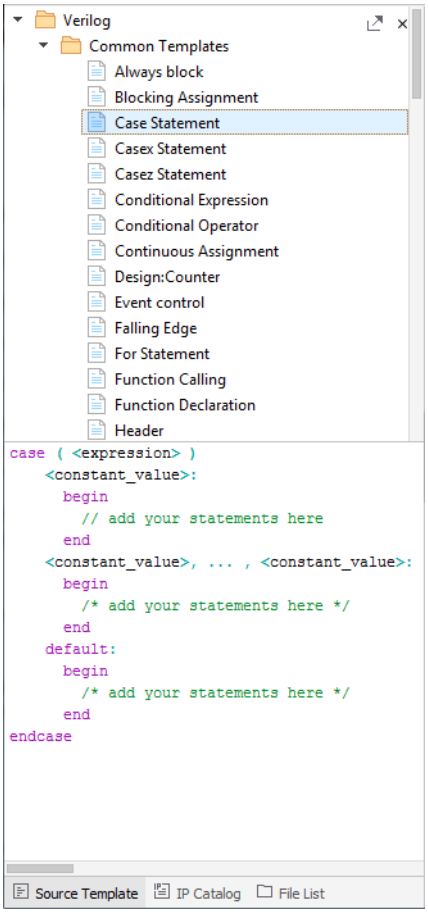
- Verilog, including common and Parameterized Module Instantiation (PMI), Primitives, Attributes, Encryption, and User Templates
- VHDL, including common, PMI, Primitives, Attributes, Encryption, and User Templates
- Constraints for LSE, including Timing and Physical constraints and User Templates

Note

For more information on PMI, refer to the Radiant software online help. See **User Guides > Entering the Design > Designing with Soft IP, Modules, and PMI > PMI or IP Catalog?**

You can simply drag any template and drop it into your source file.

Figure 22: Source Template

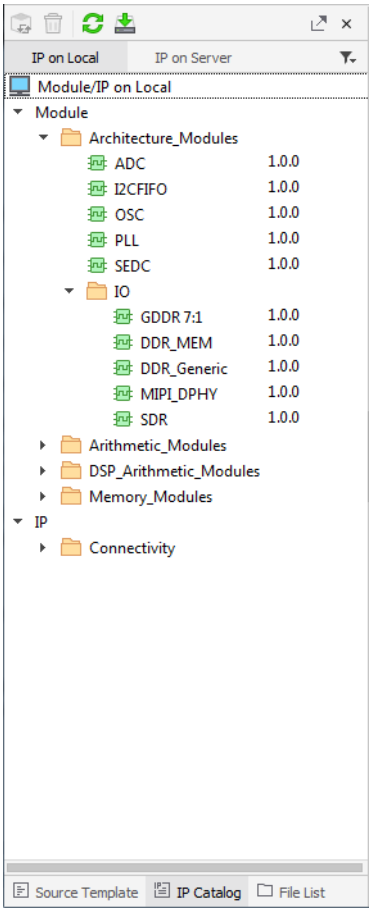


IP Catalog

IP Catalog enables you to customize a collection of functional blocks from Lattice Semiconductor. Through the IP Catalog, you can access two types of functional blocks, Modules and IP.

To access IP catalog, choose **View > Show Views > IP Catalog**, or click icon in the toolbar, or click on the **IP Catalog** tab in the bottom-left pane.

Figure 23: IP Catalog

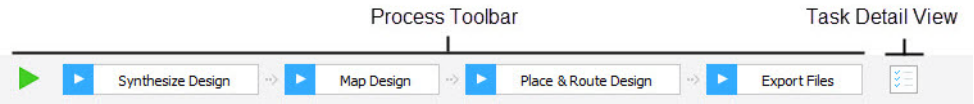


Each module is configurable with a unique set of properties. Once generated, the module or IP appears in your design's File List.

Process

A process is a specific task in the overall processing of a source or project. Typical processing tasks include synthesizing, mapping, placing, and routing. You can view the available processes for a design in the Process Toolbar.

Figure 24: Process Toolbar



The process status icons are defined as follows:

- Process in initial state (not processed)
- Process completed successfully
- Process completed with unprocessed subtasks
- Process failed

For more detail of different designs and Export Files available, see "Process Flow" on page 19.

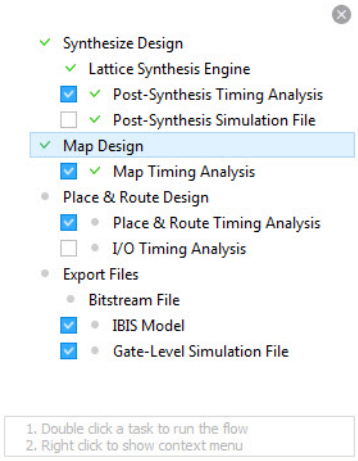
Task Detail View

Click Task Detail View to see detailed information of each process.

The default design flow processes are marked by check marks. To enable the remaining tasks, either check-mark the specific task and rerun the process step, or double-click the task's name. You can also right-click on the task to show the context menu.

Once the process has finished, the process status icon next to the task replaces the gray dot.

Figure 25: Task Detail View

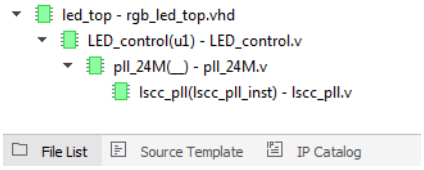


Processes are grouped into categories according to their functions. To learn more about each process, view "Design Flow Processes" on page 57.

Hierarchy

The Hierarchy view is a project view that displays the design hierarchy and is displayed by default. The hierarchical view is available when File List tab is selected.

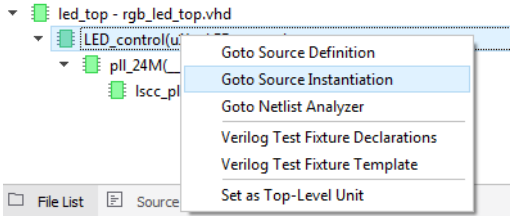
Figure 26: Hierarchy View



If you would prefer that it not open by default, simply close Hierarchy View. The next time you launch the Radiant software, the Hierarchy View will not be opened. You can open it manually by selecting it from the View > Show View menu.

Right-click any of the objects in the Hierarchy View to see the available actions.

Figure 27: Hierarchy Item Pop-up Menu



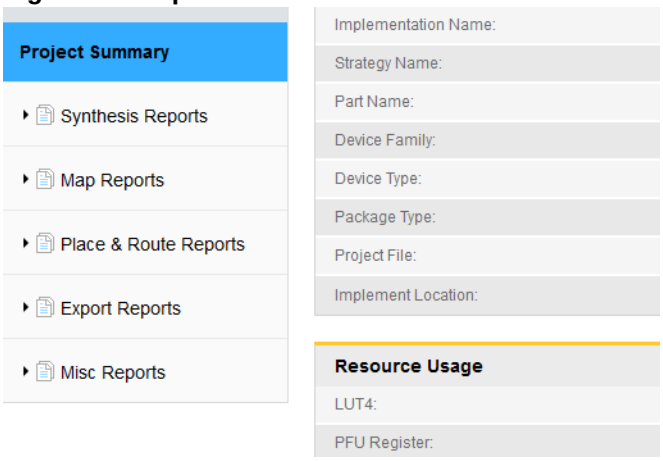
The Hierarchy view can be selected, closed, and opened.

Reports

The Reports View provides a centralized reporting mechanism in the Tools view area. The Reports View is automatically displayed and updated when processes are run. It provides a separate tab for the current implementation, enabling you to compare results quickly.

The right pane displays the report for the selected step. You can also click the icon in the toolbar.

Figure 28: Reports View



The Reports pane on the right shows the detail of the project summary and resource usage.

The Report View can be selected, closed, opened, detached, and attached with the Attach button. See “Basic UI Controls” on page 28.

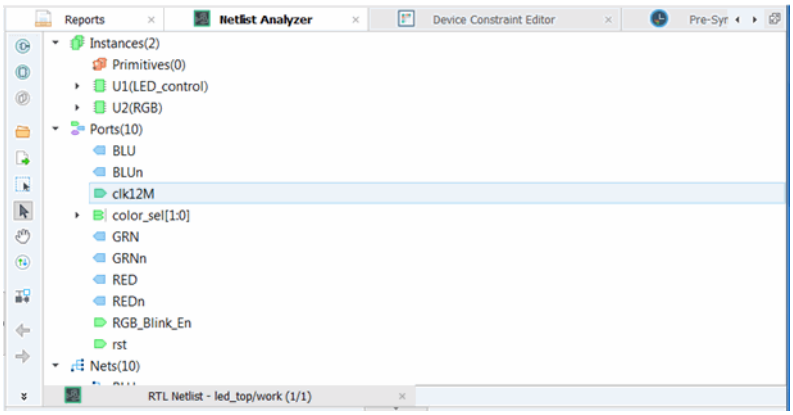
Tool Views

The Tool view area of the UI displays the tools that are currently active. Each tool that you have opened from the toolbar or the Tools menu is displayed. The Reports and Start page, which can be opened from the toolbar or the Windows menu, are also displayed. When multiple tools are active, the display can be controlled with the tab group functions in the Window menu. See “Common Tasks” on page 36 for more information on tab group functions.

Each tool view is specific to its tool and can contain additional toolbars, multiple panes, or multiple windows controlled by additional tabs. See “Working with Tools and Views” on page 71 for descriptions of each tool and view, plus details on controlling their display.

The Tool views can be selected, closed, opened, detached, and attached using the Attach button. See “Basic UI Controls” on page 28

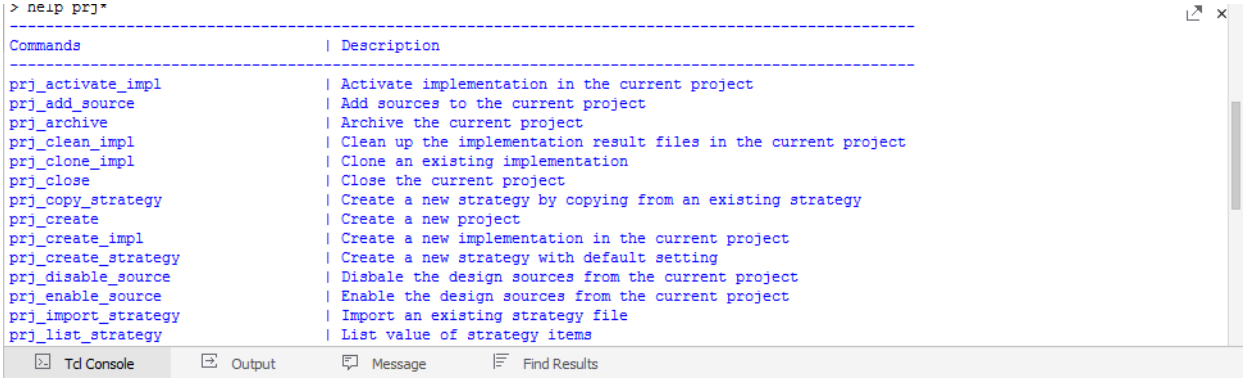
Figure 29: Tool View Tab Title



Tcl Console

The Tcl Console is an integrated console for Tcl scripting. You can enter Tcl commands in the console to control all of the functionality of the Radiant software. Use the Tcl help command (help <*tool_name*>*) to display a list of valid extended Tcl commands.

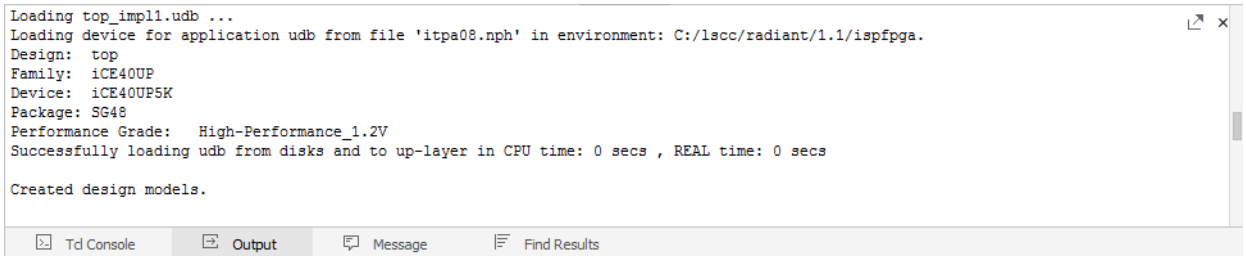
Figure 30: Tcl Console



Output

The Output View is a read-only area where tool output is displayed.

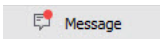
Figure 31: Output View



Message

There are three message types available:

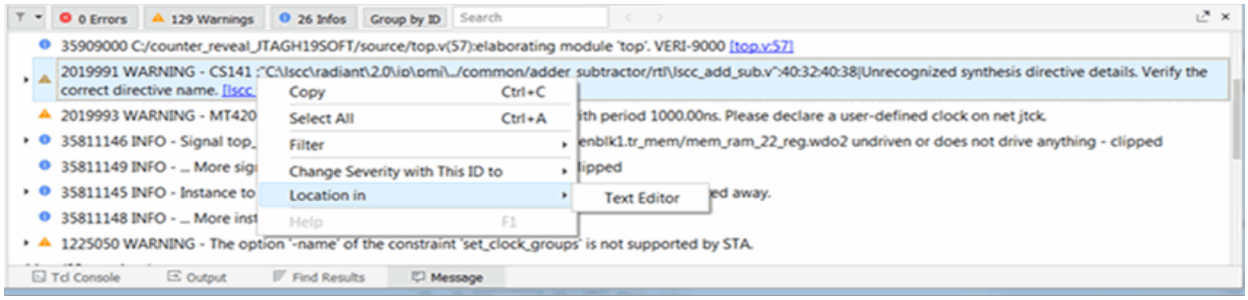
- Errors are displayed in red.
- Warnings are displayed in orange.
- General Information is displayed in blue.



A red dot in the Message tab provides a visual notification that a new message/warning was received. Once you view the notification, the dot disappears.

Right-clicking a message provides a menu of commands, including **Location in > Text Editor**, which opens the source file in the Source Editor and highlights the location of the problem.

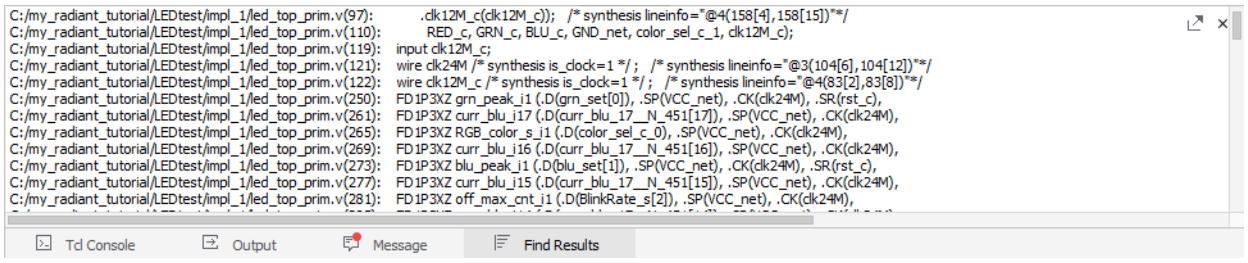
Figure 32: Message Display



Find Results

The **Edit > Find in Files** command enables you to search for information in the files within your project directory. The search results are then displayed in the Find Results view.

Figure 33: Find Results View



Common Tasks

The Radiant software UI controls many tools and processes. The following sections describe some of the more commonly performed tasks.

Controlling Views

All of the views in the Radiant software are controlled in a similar manner, even though the information they contain varies widely. Here are some of the most common operations:

- Open – Use the **View > Show Views** menu selections or right-click in the menu or toolbar areas to select a view from the pop-up menu.
- Select – If a view is already open you can select its tab to bring it to the front.
- Detach – Click the detach button in the upper right corner of the view.
- Attach – Click the attach button in the upper right corner of the view.
- Move – Click and hold a view's tab, and then drag and drop the view to a different position among the open views.

Using a Tab Group You can use the Window menu to split off a view and control it as a separate tab group. This allows you to examine two open views side by side. The controls work as follows:

- Split Tab Group – displays two views side by side. For more information, refer to Figure 34.
- Move to Another Tab Group – moves the selected tab to the other tab group.

Figure 34: After Split Tab Group Command Used on Physical Designer

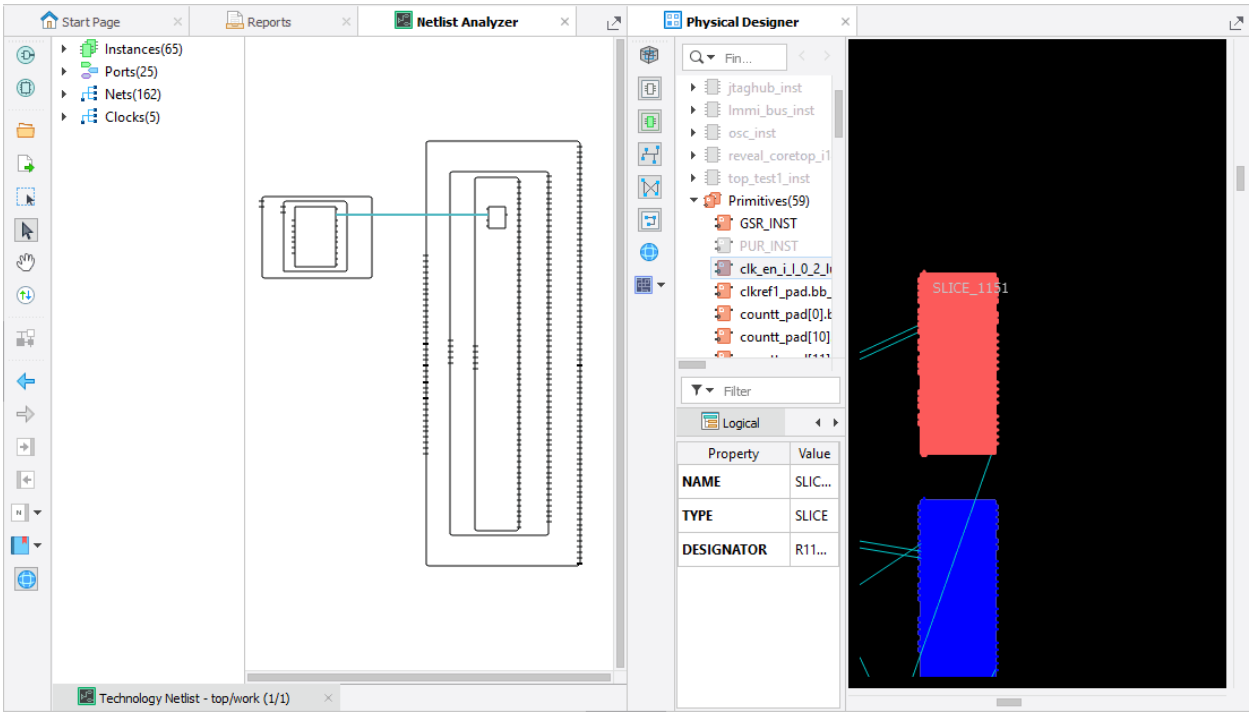
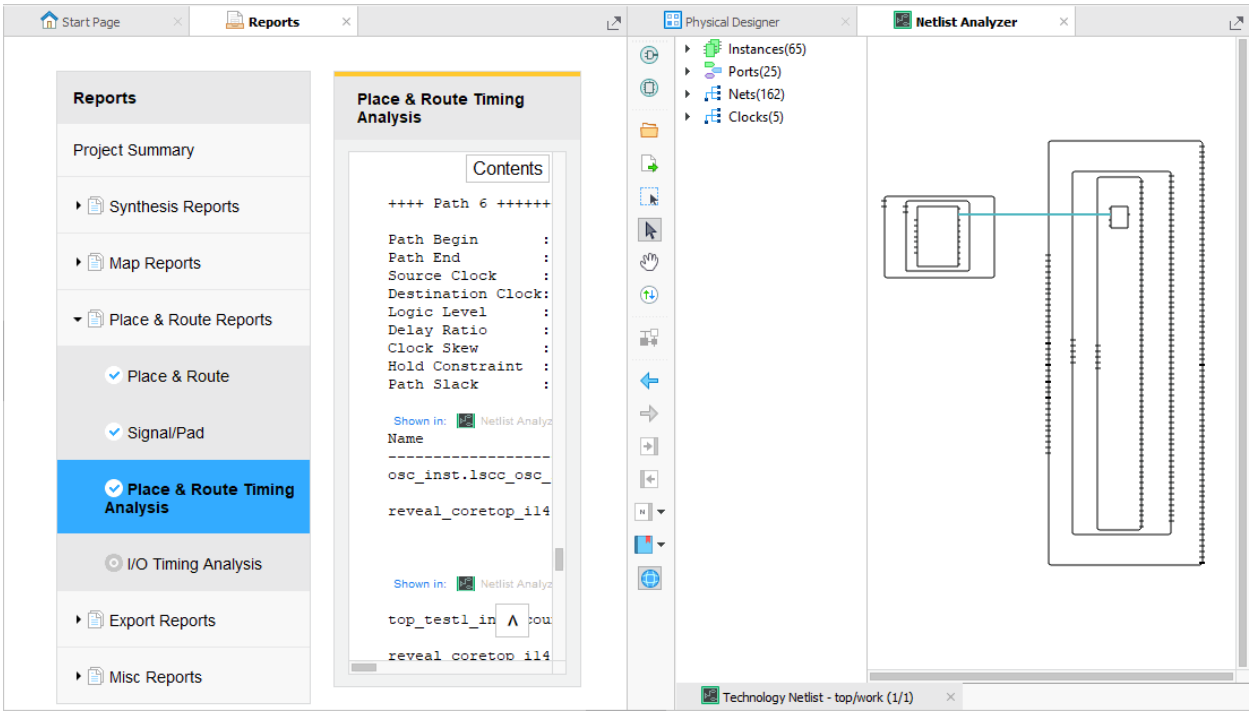


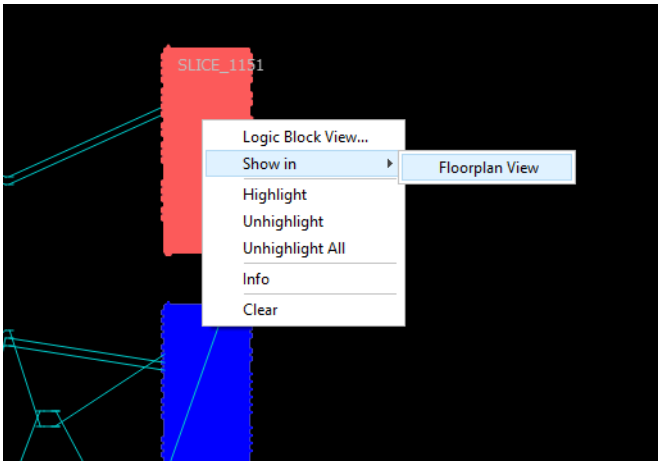
Figure 35: After Move to Another Tab Group Used on Netlist Analyzer



Cross-Probing

It is possible to select a data object in one view and see that same data object in a different view or views. Right-click an object to see if cross-probing is available. If it is, you will see a **Show In** sub-menu with the available views listed. If you select a view that is not yet open, the Radiant software will open it automatically. Cross-probing is available between Floorplan View and Physical View of Physical Designer, and from Netlist Analyzer to Physical Designer.

Figure 36: Show In Menu from Physical View of Physical Designer



During the Radiant flow, various timing analyses and reports are created. You can view a specific path in Netlist Analyzer, Physical Designer's Floorplan View, and Physical Designer's Physical View. This allows for flexibility and reduced debugging effort.

NOTE

Cross-probing to Netlist Analyzer is available only if the selected synthesis tool is LSE.

In the Reports tab, view any timing analysis report and identify a path to view. If cross-probing is available, the specific icon tools become visible, as shown in the following figure.

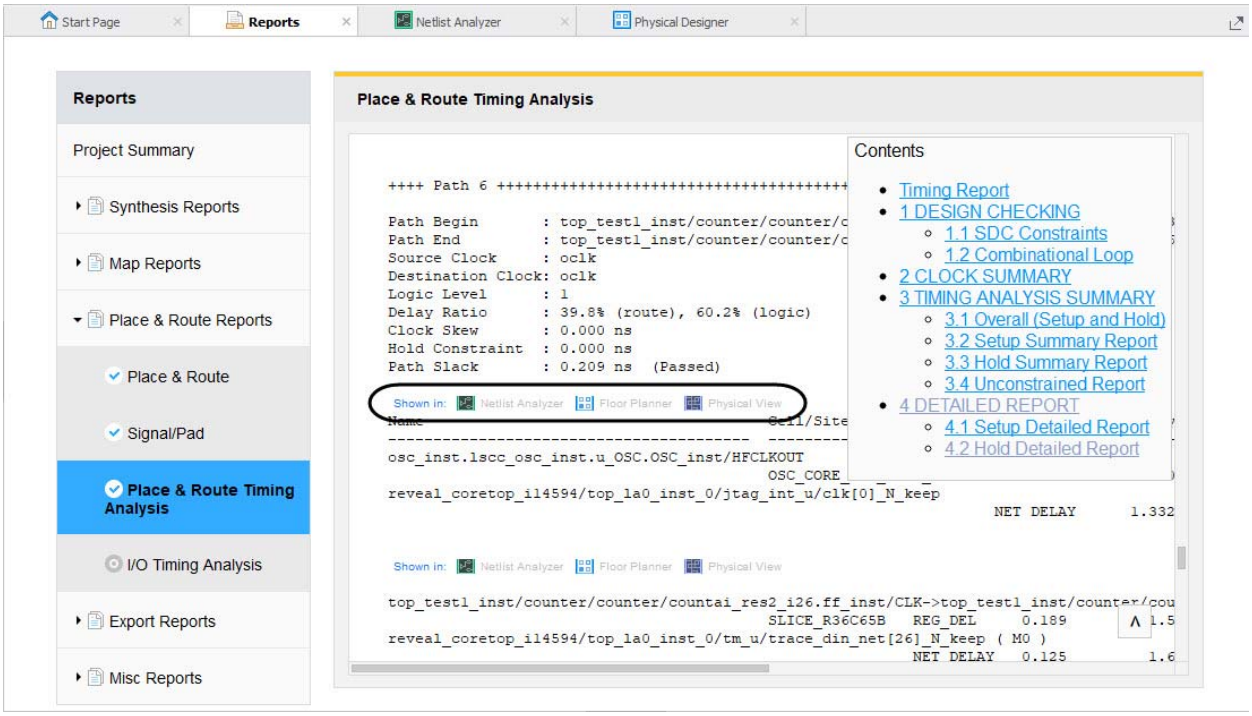
Figure 37: Available tools for Path Cross-Probing

Click on an icon and the tool opens with the selected path.

In some cases, the tool is unable to find the path. The message "Can't show the schematic of this timing path." appears. In an encrypted design, in some cases, cross-probing is not available. The message "Cannot open encrypted design." appears.

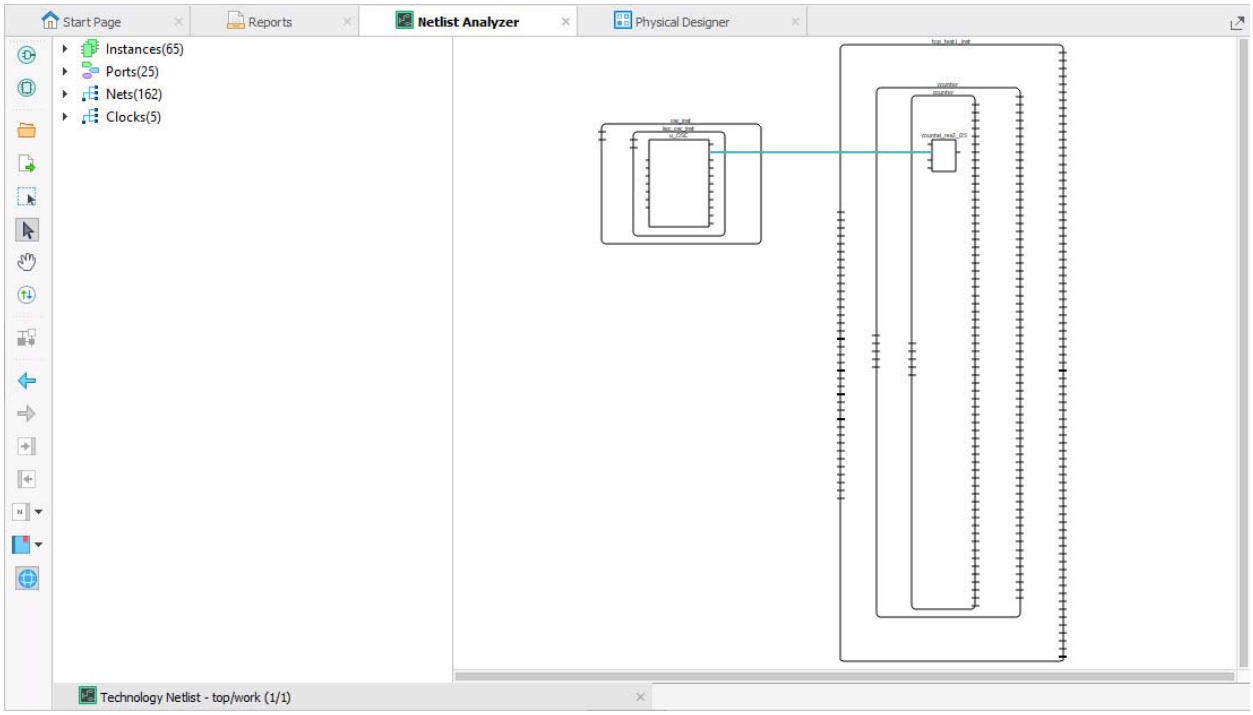
The following figures show cross-probing a path from the Place & Route Timing Analysis report to Netlist Analyzer, Physical Designer's Floorplan View, and Physical Designer's Physical View.

Figure 38: Path Cross-Probing in Reports



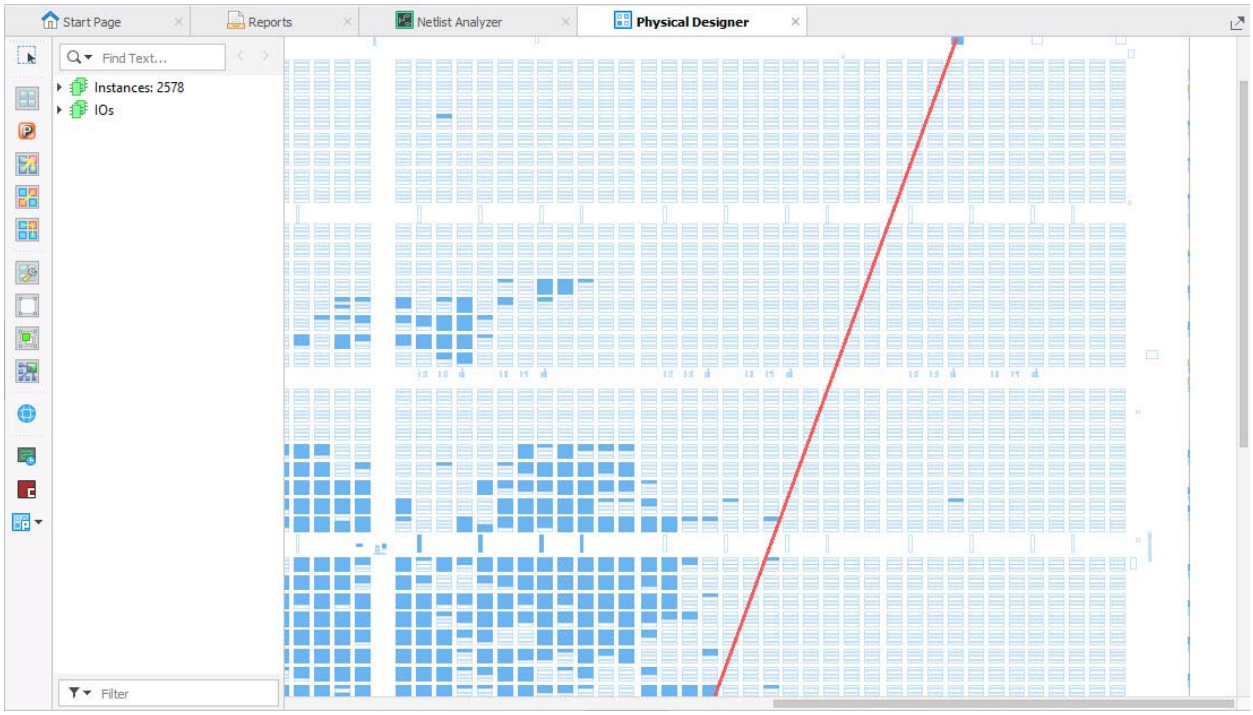
By clicking on the Netlist Analyzer icon, you can preview the data path in Netlist Analyzer.

Figure 39: Path Cross-Probing in Netlist Analyzer



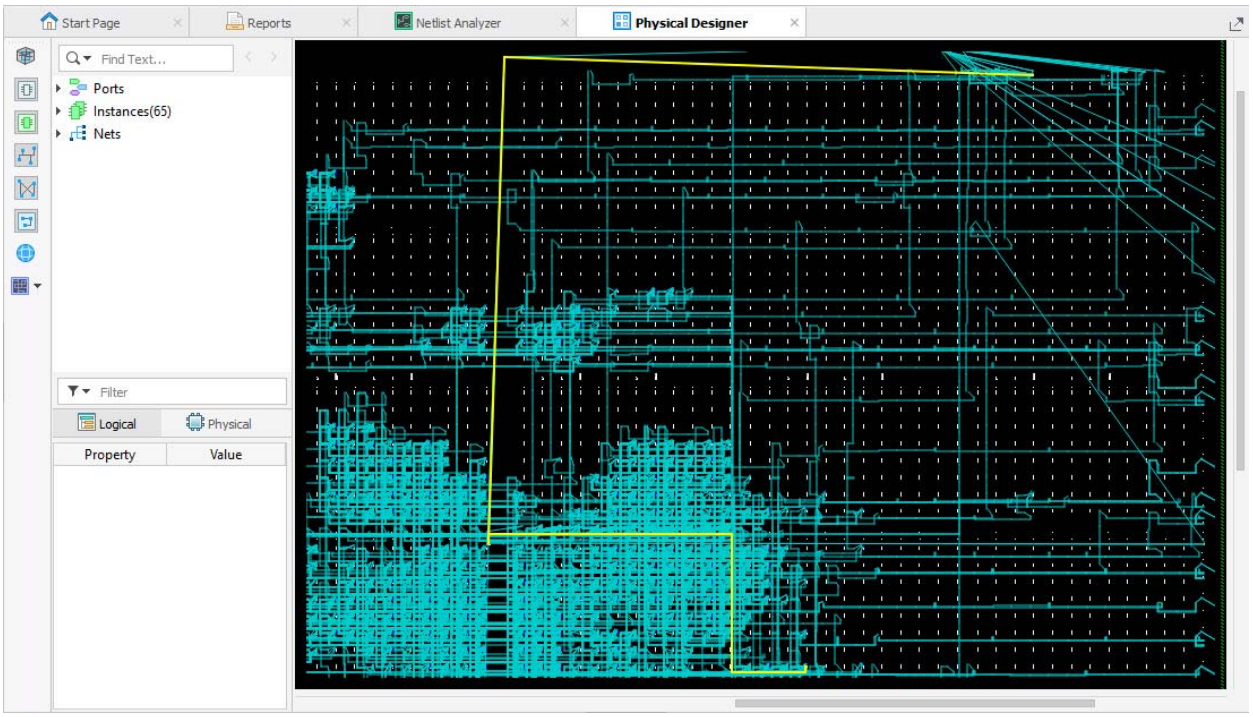
Similarly, by clicking on the Floor Planner icon, you can easily view the same path in Floorplan View.

Figure 40: Path Cross-Probing in Floorplan



The same path is viewable in Physical View by clicking on the Physical View icon in a timing report.

Figure 41: Path Cross-Probing in Physical Designer: Physical View



Chapter 5

Working with Projects

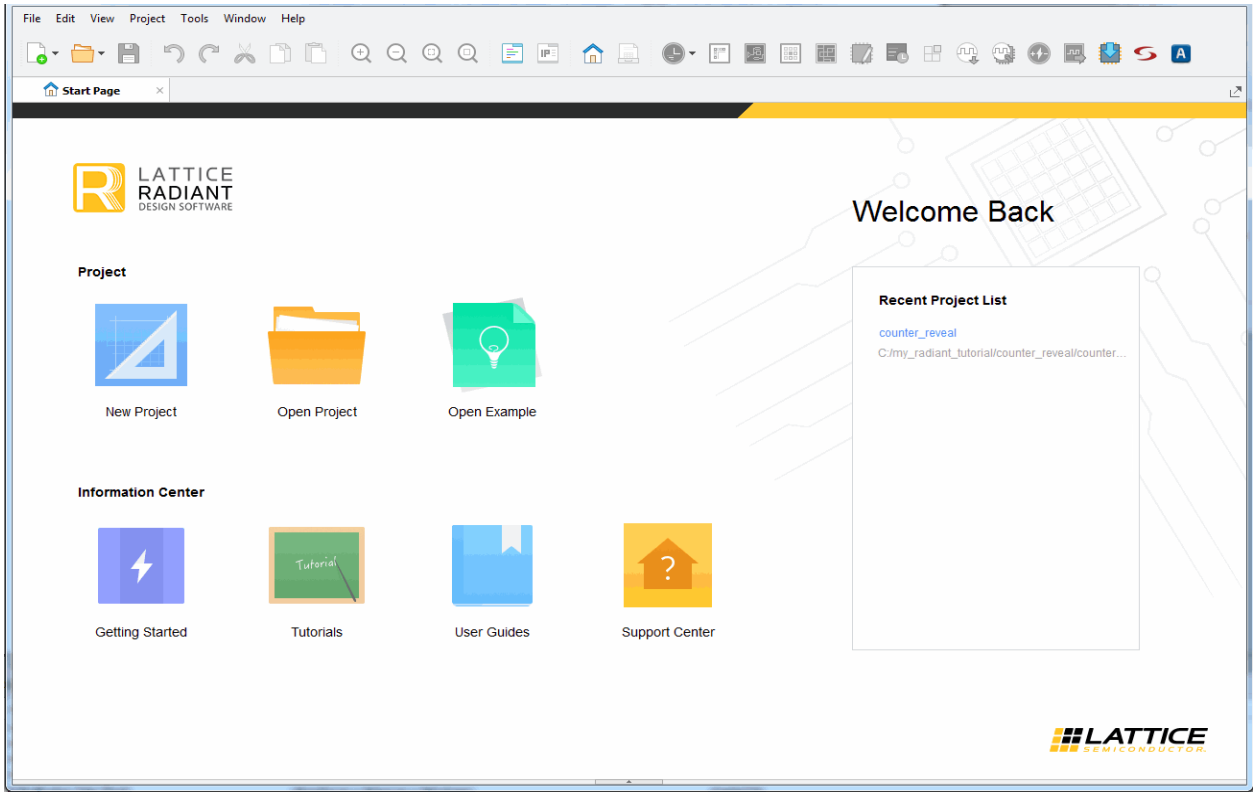
This chapter covers projects and their elements. Implementations and strategies are explained and some common project tasks are shown.

Overview

A project is the top organizational element in the Radiant software design environment. Projects consist of design, constraint, configuration and analysis files. Only one project can be open at a time, and a single project can include multiple design structures and tool settings.

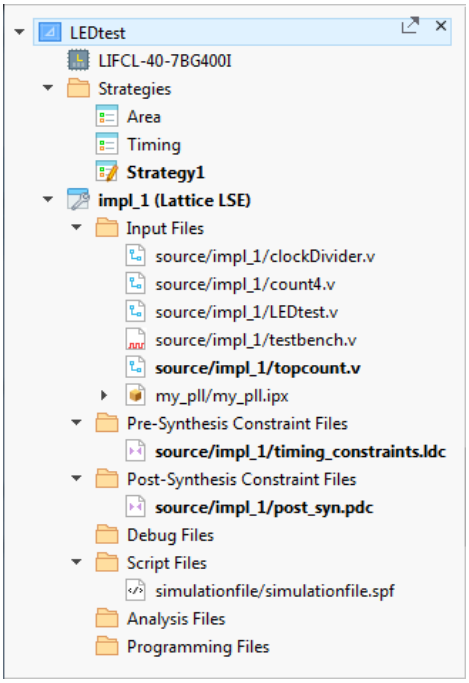
You can create, open, or import a project from the Start Page. Refer to "Getting Started" on page 10 for instructions on creating a new project.

Figure 42: Default Start Page



The File List view shows a project and its main elements.

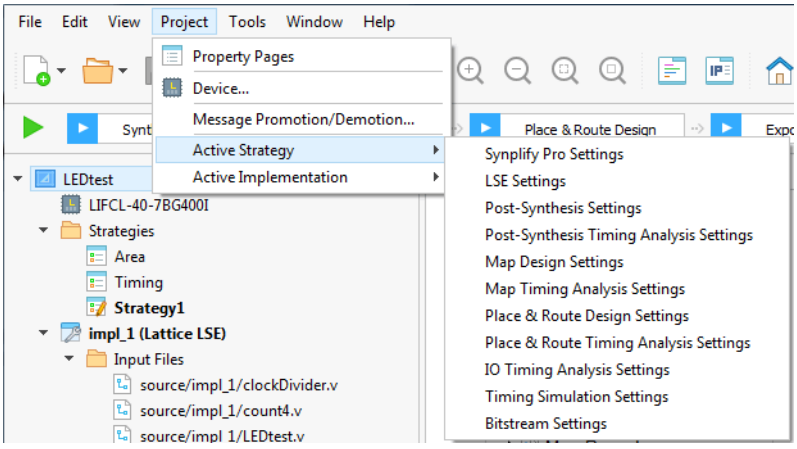
Figure 43: Project Files in File List



The Project menu commands enable you to do the following:

- Examine the project properties.
- Change the target device.
- Change the severity level of warning messages.
- Set the synthesis tool.
- Show the active strategy tool settings.
- Set the top level design unit.

Figure 44: Project Menu



Implementations

An implementation is the structure of a design and can be thought of as *what* is in the design. For example, one implementation might use inferred memory while another implementation uses instantiated memory. Implementations also define the constraint and analysis parameters for a project.

There can be multiple implementations in a project, but only one implementation can be active at a time. And there must be one active implementation. Every implementation has an associated active strategy. Strategies are a shared pool of resources for all implementations and are discussed in the next section. An implementation is created whenever you create a new project.

Implementations consist of the following files:

- Input files
- Pre-Synthesis constraint files
- Post-Synthesis constraint files
- Debug files
- Script files
- Analysis files
- Programming files

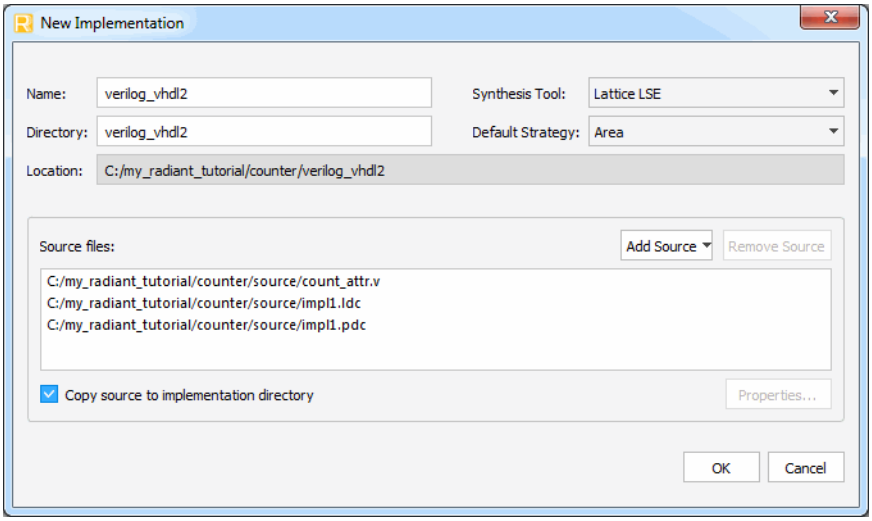
Adding Implementations

To add a new implementation to an existing project:

1. Right-click the project name in the File List project view.

Select **Add > New Implementation**. In the New Implementation dialog box, you can set the implementation name, directory, default strategy, and add source files. When you select **Add Source** you have a choice of browsing for the source files or using a source from an existing implementation.

Figure 45: New Implementation



Notice that you have the option to “Copy source to implementation directory.” If this option is selected, the source files will be copied from the existing implementation to the new implementation, and you will be working with different source files in the two implementations. If you want the two implementations to share the same source files and stay in sync, make sure that this option is not selected.

To make an implementation active, right-click its name in the File List and choose **Set as Active Implementation**.

To add a file to an implementation, right-click the implementation name or any file folder in the implementation and choose **Add > New File** or **Add > Existing File**.

Cloning Implementations

To clone an implementation:

1. In the File List view, right-click on the name of the implementation that you want to copy and choose **Clone Implementation**.

   The Clone Implementation dialog box opens.

2. In the dialog box, enter a name for the new implementation. This name also becomes the default name for the folder of the implementation.
3. Change the name of the implementation’s folder in the Directory text box, if desired.
4. Decide how you want to handle files that are outside of the original implementation directory. Select one of the following options:

   - **Continue to use the existing references**

     The same files will be used by both implementations.

- **Copy files to new implementation source directory**

  The new implementation will have its own copies that can be changed without effecting the original implementation.

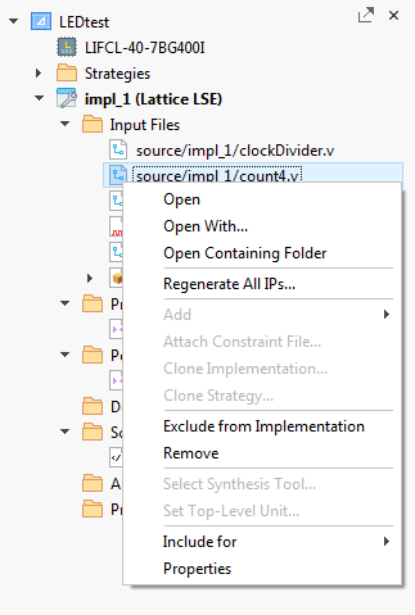
5. The Synthesis Tool text box specifies the currently selected synthesis tool. Go to **Project** > **Active Implementation** > **Select Synthesis Tool** to update your selection.
6. The Default Strategy text box specifies the currently selected default strategy.
7. Click **OK**.

Input Files

Input files are the design source files for the project. Input files can be any combination of Verilog, SystemVerilog, and VHDL.

Right-click an input file name to open a pop-up menu of possible actions for that file.

Figure 46: Input File Actions



You can use the “Include for” commands to specify that a source file be included for both synthesis and simulation, synthesis only, or simulation only.

Pre-Synthesis Constraint Files

Synopsys timing constraints are specified in the new .fdc file format. Legacy .sdc formats are still supported in the Radiant software and Synopsys has provided a script called sdc2fdc, which does a one-time conversion of .sdc files to the new .fdc format. More information about this script can be found in the Synplify Pro release notes.

An .sdc or .fdc file can be added to an implementation if the selected synthesis tool is Synplify Pro. If the selected synthesis tool is the Lattice Synthesis Engine (LSE), a Lattice design constraint (.ldc) synthesis file can be added. Constraints in the .ldc file use the Synopsys constraint format.

An implementation can have multiple synthesis constraint files. Only one synthesis constraint file can be active at a time. Unlike Post-Synthesis constraints, a synthesis constraint file must be set as active by the user.

Post-Synthesis Constraint Files

Post-Synthesis constraint files (.pdc) contain both timing and non-timing constraint .pdc source files for storing logical timing/physical constraints. Constraints that are added using the Device Constraint Editor are saved to the active .pdc file. The active post-synthesis design constraint file is then used as input for post-synthesis processes.

An implementation can have multiple .pdc files, but only one can be active at a time.

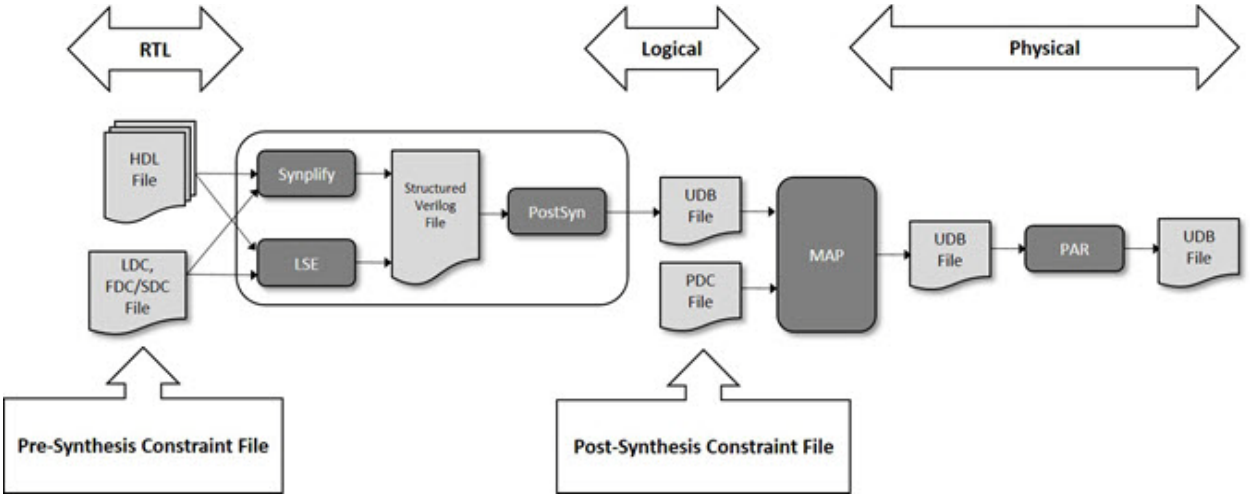
Figure 47: Sample .pdc File

impl1.pdc

```
1  ldc_set_location -site P5 [get_ports clk]
2  #ldc_create_region -name RgnA -site R8C2D -width 18 -height 12
3  #ldc_create_group -name UgrpA [get_cells counter1]
4  #ldc_set_location -region RgnA [ldc_get_groups UgrpB]
5  #ldc_create_group -name UgrpC -bbox {2 3} [get_cells counter3]
6  #ldc_set_location -site R8C25D [ldc_get_groups UgrpC]
7  ldc_set_port -iobuf {IO_TYPE=LVCMOS25} [get_ports {count2t[2]}]
8  ldc_set_port -iobuf {IO_TYPE=LVCMOS25} [get_ports {count2t[3]}]
9  ldc_set_port -iobuf {IO_TYPE=LVCMOS25} [get_ports {count2t[0]}]
10 ldc_set_port -iobuf {IO_TYPE=LVCMOS25} [get_ports {count2t[1]}]
11 ldc_set_port -iobuf {IO_TYPE=LVCMOS25} [get_ports {count3t[0]}]
12 ldc_set_port -iobuf {IO_TYPE=LVCMOS25} [get_ports {count3t[1]}]
13 ldc_set_port -iobuf {IO_TYPE=LVCMOS25} [get_ports {count3t[2]}]
14 ldc_set_port -iobuf {IO_TYPE=LVCMOS25} [get_ports {count3t[3]}]
15 ldc_set_port -iobuf {IO_TYPE=LVCMOS25} [get_ports {count3t[4]}]
16 ldc_set_port -iobuf {IO_TYPE=LVCMOS25} [get_ports {count3t[5]}]
17 ldc_set_port -iobuf {IO_TYPE=LVCMOS25} [get_ports {count3t[6]}]
18 ldc_set_port -iobuf {IO_TYPE=LVCMOS25} [get_ports {count3t[7]}]
19 #ldc_set_location -site 35 [get_ports clk1]
20 create_clock -period 15 [get_nets counter1/clk1_c]
21
```

Figure 48 shows a high-level flow of how constraints from multiple sources can be used and modified in the Radiant software.

Figure 48: Radiant software Constraints Flow Chart



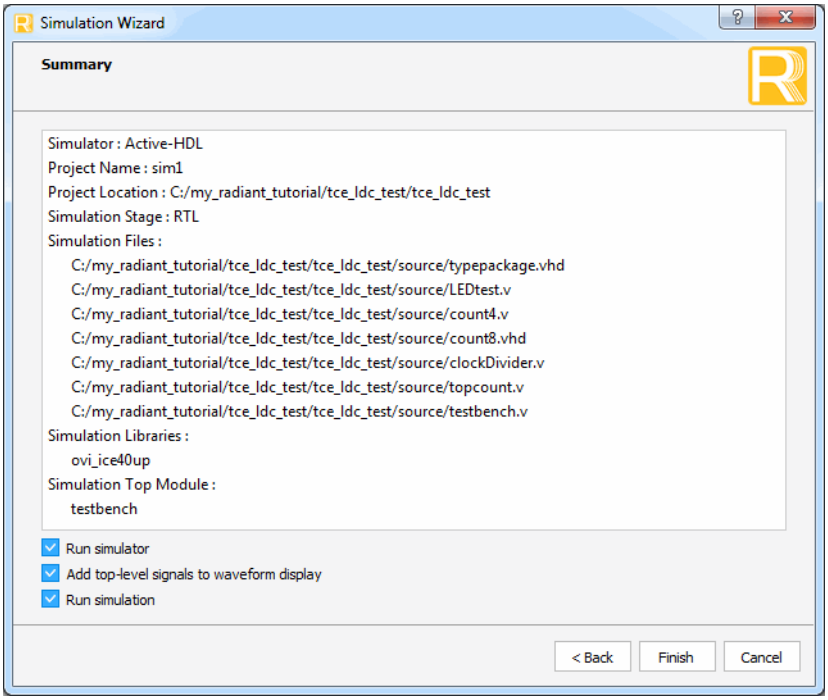
Debug Files

The files in the Debug folder are project files for Reveal Inserter. They are used to insert hardware debug into your design. There can be multiple debug files, and one can be set as active. To insert hardware debug into your design, right-click a debug file name and choose **Set as Active Debug File** from the pop-up menu. The debug file name becomes bold, indicating that it is active. It is not required to have an active debug file.

Script Files

The Script Files folder contains the scripts that are generated by the Simulation Wizard. After you run the Simulation Wizard, the steps are stored in a simulation project file (.spf), which can be used to control the launching of the simulator.

Figure 49: Simulation Script File



Analysis Files

The Analysis Files folder contains Power Calculator files (.pcf). The folder can contain multiple analysis files, and one (or none) can be set as active. The active or non-active status of an analysis file affects the behavior of the associated tool view.

Programming Files

Programming files (.xcf) are configuration scan chain files used by the Radiant Programmer for programming devices. The .xcf file contains information about each device, the data files targeted, and the operations to be performed.

An implementation can have multiple .xcf files, but only one can be active at a time.The file must be set as active by the user.

Strategies

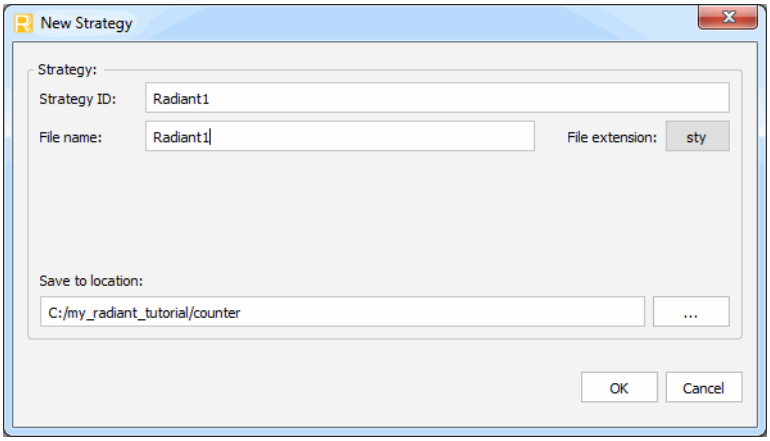
Strategies are collections of all the implementation-related tool settings in one convenient location. Strategies can be thought of as recipes for how the design will be implemented. An implementation defines *what* is in the design,

and a strategy defines *how* that design will be run. There can be many strategies, but only one can be active at a time. There must be one active strategy for each implementation.

The Radiant software provides two predefined strategies: Area and Timing. It also enables you to create customized strategies. Predefined strategies cannot be edited, but they can be cloned, modified, and saved as customized user strategies. Customized user strategies can be edited, cloned, and removed. All strategies are available to all of the implementations, and any strategy can be set as the active one for an implementation.

To create a new strategy from scratch, choose **File > New > Strategy**. In the New Strategy dialog box, enter a name for the new strategy. Specify a file name for the new strategy and choose a directory to save the strategy file (.sty).

Figure 50: Creating a New Strategy from Scratch

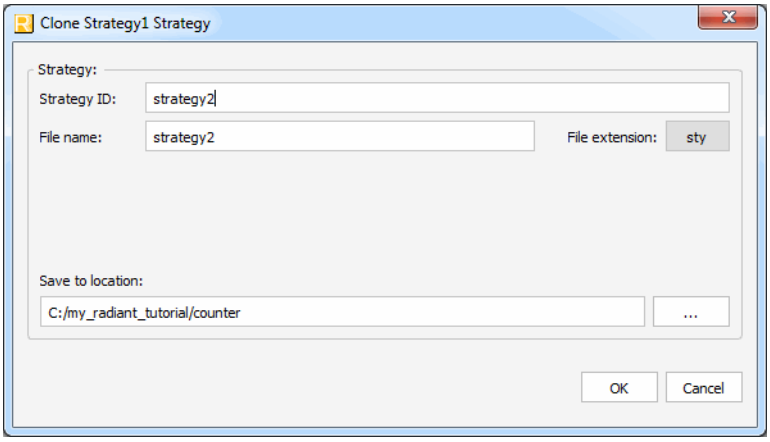


The new strategy is with all the default settings of the current design. You can modify its settings in the Strategies dialog box.

If you want to save the strategy changes to your current project, choose **File > Save Project** from the Radiant software main window.

To create a new strategy from an existing one, right-click the existing strategy and choose **Clone <*strategy name*> Strategy**. Set the new strategy's ID and file name.

Figure 51: Cloning to Create a New Strategy



To make a strategy active, right-click the strategy name and choose **Set as Active Strategy**.

To change the settings in a strategy:

1. Double-click the strategy name in the File List view
2. Select the option type to modify
3. Double-click the Value of the option to be changed

The default values are displayed in plain blue text. Modified values are displayed in italic bold text.

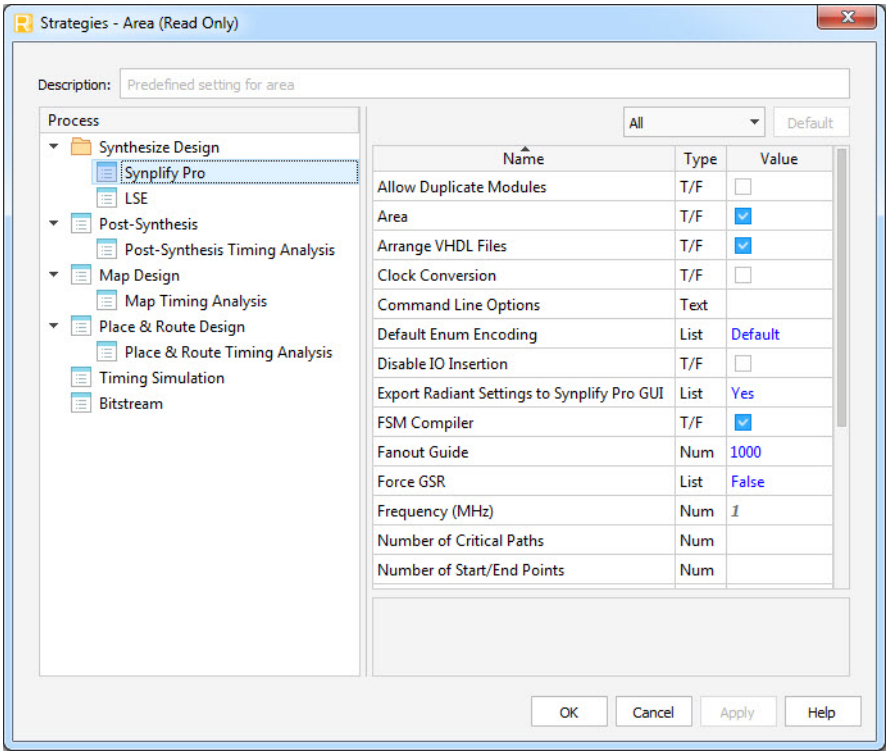
Strategies are design data independent and can be exported and used in multiple projects.

Area

The Area strategy is a predefined strategy for area optimization. Its purpose is to minimize the total logic gates used while enabling the tight packing option available in Map.

Applying this strategy to large and dense designs might cause difficulties in the place and route process, such as longer time or incomplete routing.

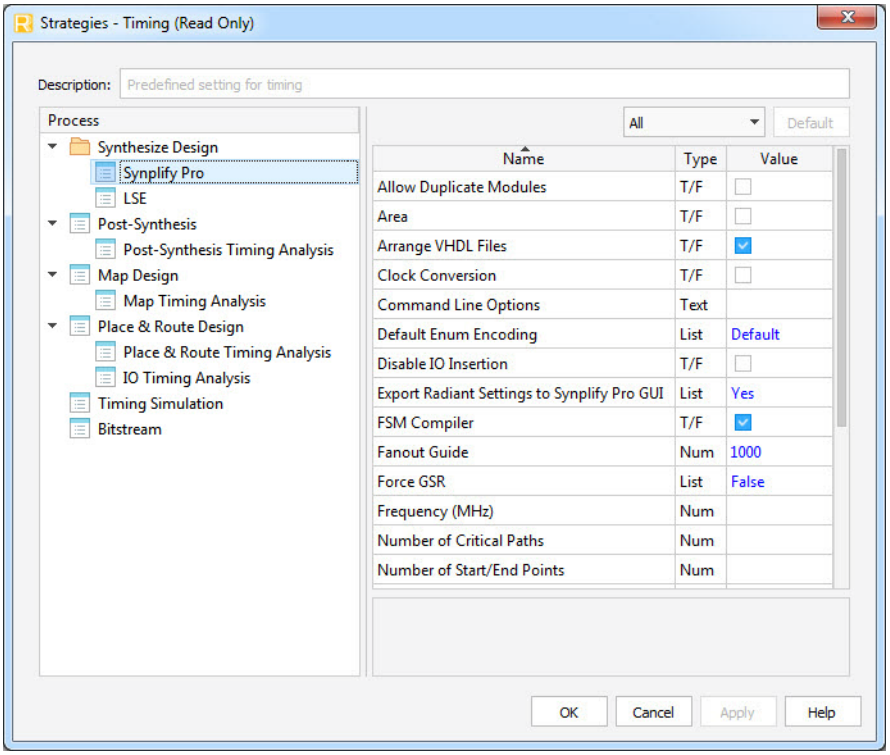
Figure 52: Area Predefined Strategy



Timing

The Timing strategy is a predefined strategy for timing optimization. Its purpose is to achieve timing closure. The Timing strategy uses a very high effort level in placement and routing. Use this strategy if you are trying to reach the maximum frequency on your design. If you cannot meet your timing requirements with this strategy, you can clone it and create a customized strategy with refined settings for your design. This strategy might increase your place-and-route run time compared to the Area strategy.

Figure 53: Timing Predefined Strategy



User-Defined

You can define your own customized strategy by cloning and modifying any existing strategy. You can start from either a predefined or a customized strategy.

Common Tasks

Working with projects includes many tasks, including: creating the project, editing design files, modifying tool settings, trying different implementations and strategies, and saving your data.

Creating a Project

See “Creating a New Project” on page 11 for step-by-step instructions.

Changing the Target Device

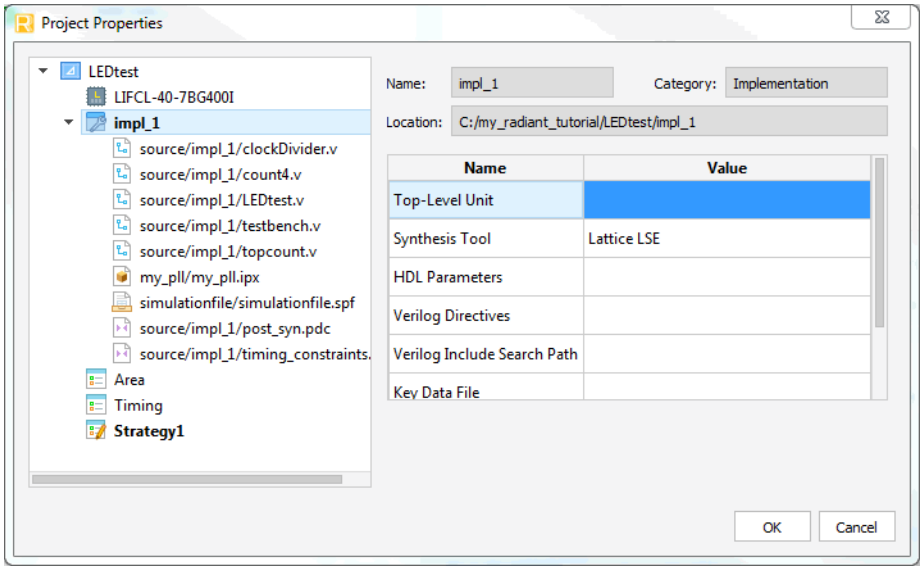
There are two ways to access the Device Selector dialog box for changing the target device:

- Double-click the device in the project File List view or right-click it and choose **Edit**.
- Choose **Project > Device**.

Setting the Top Level of the Design

If multiple top levels exist in the hierarchy of your HDL source files, you will need to set the top-level design unit. After generating the hierarchy, choose **Project > Active Implementation > Set Top-Level Unit**. Alternatively, right-click the implementation and choose this command from the pop-up menu.

Figure 54: Top-Level Design Unit



In the Project Properties dialog box, select **Value** next to **Top-Level Unit** and select the desired top level from the list.

You can also use the Hierarchy View to set the top-level. Right-click a level you want to be the top-level in the Hierarchy View and choose **Set Top-Level Unit**.

Editing Files

You can open any of the files for editing by double-clicking or by right-clicking and choosing **Open** or **Open with**.

Saving Project Data

In the File menu are the following selections for saving your design and project data:

- Save – saves the currently active item.
- Save As – saves the active item using a different file name.
- Save All – saves all changed documents.
- Save Project – saves the current project.
- Save Project As – saves the active project using a different project name.
- Archive Project – creates a zip file of the current project in a location you specify.

Chapter 6

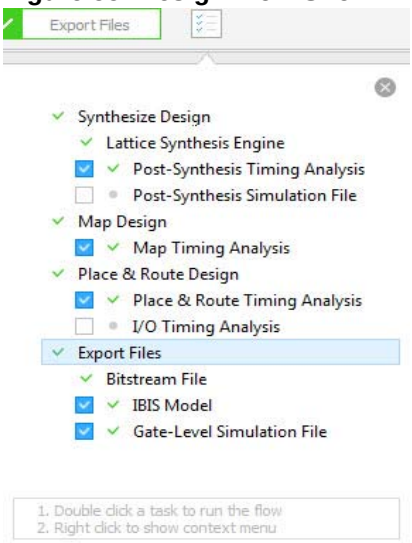
Radiant Software Design Flow

This chapter describes the design flow in the Radiant software. Running processes and controlling the flow for alternate what-if scenarios are explained.

Overview

The FPGA implementation design flow in the Radiant software provides extensive what-if analysis capabilities for your design. The design flow is displayed in the Task Detail View at the right end of the Process Toolbar.

Figure 55: Design Flow Shown in Task Detail View



Design Flow Processes

The design flow is organized into discrete processes, where each step allows you to focus on a different aspect of the FPGA implementation.

Synthesize Design This process runs the selected synthesis tool (Lattice Synthesis Engine is the default) in batch mode to synthesize your HDL design.

- Synthesis Tool - identifies the selected synthesis tool, Lattice Synthesis Engine or Synplify Pro.
- Post-Synthesis Timing Analysis - generates timing analysis files.
- Post-Synthesis Simulation File - generates a post-synthesis netlist file <*file_name*>_syn.vo used for Post-Synthesis Simulation.

Map Design This process maps the design to the target FPGA and produces a mapped Unified Database (.udb) design file. Map Design converts a design's logical components into placeable components.

- Map Timing Analysis - generates timing analysis files.

Place & Route Design This process takes mapped physical design files and places and routes the design. The output can be processed by the design implementation tools. Timing analysis files can also be generated.

- Place & Route Timing Analysis - generates timing analysis files.
- I/O Timing Analysis - generates I/O timing analysis files.

Export Files This process generates the IBIS, simulation, and programming files that you have selected for export:

- Bitstream File – generates a configuration bitstream (bit images) file, which contains all of the design's configuration information that defines the internal logic and interconnections of the FPGA, as well as device-specific information from other files.
- IBIS Model – generates a design-specific I/O Buffer Information Specification model file (.ibs). IBIS models provide a standardized way of representing the electrical characteristics of a digital IC's pins (input, output, and I/O buffers).
- Gate-Level Simulation File – generates a Verilog netlist of the routed design that is back annotated with timing information. This generated .vo file enables you to run a timing simulation of your design.

The files for export can also be generated separately by double-clicking each one.

Running Processes

You can perform the following actions for each step in the process flow:

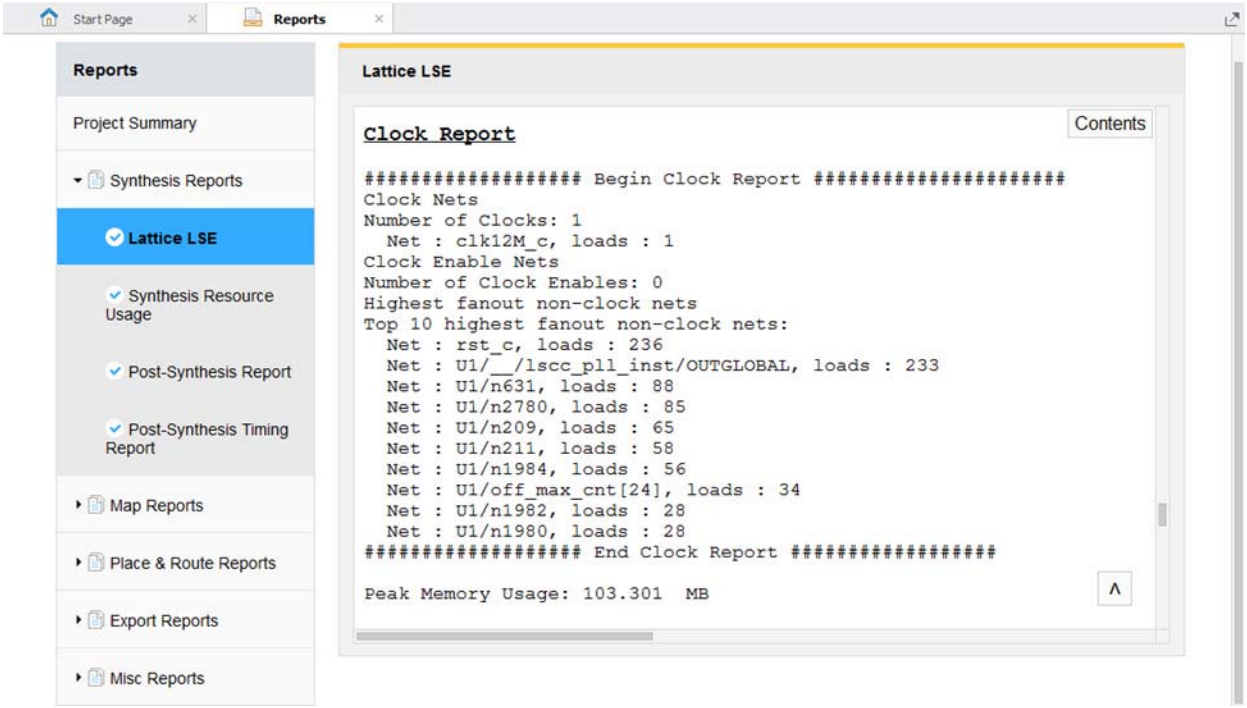
- Run – runs the process, if it has not yet been run.
- Force Run– reruns a process that has already been run.

- Force Run From Start – reruns all processes from the start to the selected process.
- Stop – stops a running process.
- Clean Up Process – clears the process state and puts a process into an initial state as if it had not been run.

The state of each process step is indicated with an icon to the left of the process. The process status icons description is described in “Process” on page 31

The Reports View displays detailed information about the process results, including the last process run. The Messages section shows warning and error messages and allows you to filter the types of messages that are displayed. See “Reports” on page 33.

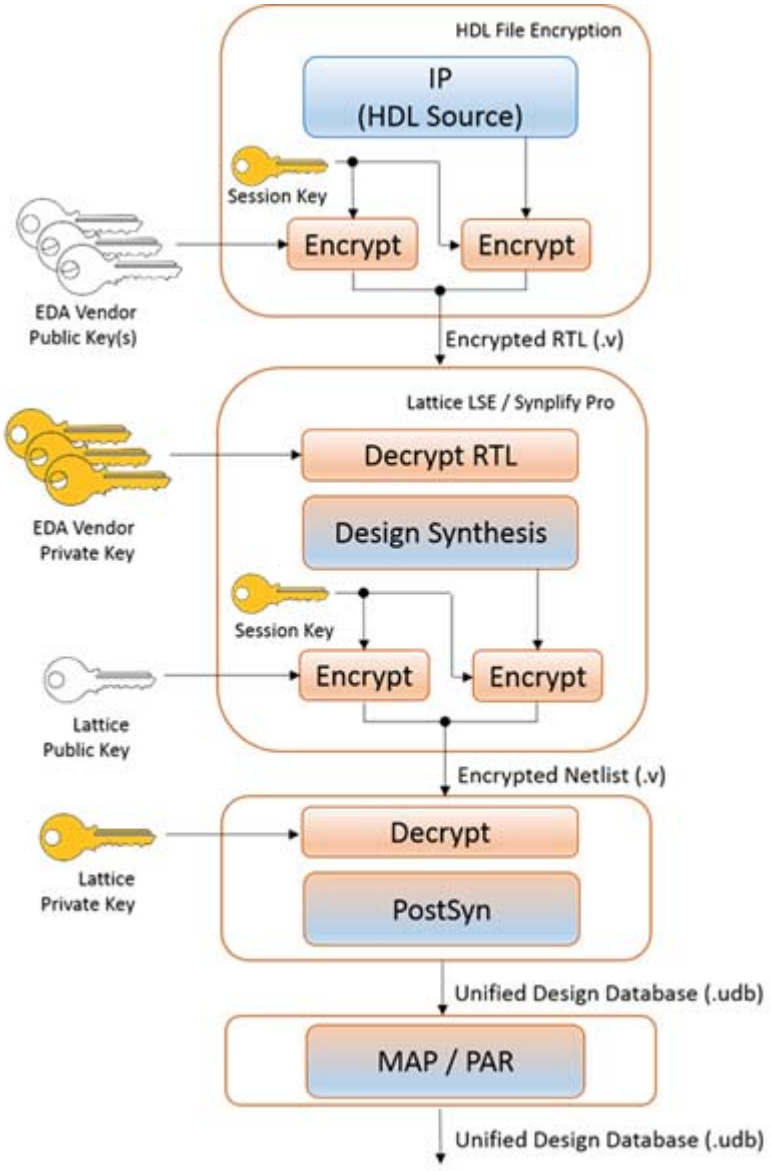
Figure 56: Reports View of Last Process Run



IP Encryption Flow

IP encryption flow enables you to protect your IP design. Following the industry standard, the Radiant software, through the IP encryption flow, allows the partnership between the IP Vendor, supported EDA vendor, and Lattice Semiconductor.

Figure 57: IP Encryption Flow



The encryption flow uses symmetric and asymmetric encryption methods to maximize the design security. The symmetric method only involves a single symmetric key for both, encryption and decryption. The asymmetric method involves the public-private key pair. The public key is published by a vendor and is used by the Radiant software. The private key is never revealed to the public.

The Radiant software supports these cryptographic algorithms:

- AES-128/AES-256: symmetric algorithm used to encrypt the content of the HDL source file.
- RSA-2048: asymmetric algorithm used to obfuscate a key used in HDL file encryption. The RSA is defined by the public-private key pair. You must be familiar with both keys in order to perform RSA decryption.

HDL File Encryption Flow

The current software version supports encryption of a single HDL source file per a single command.

The overall HDL file encryption flow is summarized in these steps:

- The source file of the IP design is AES encrypted using a symmetric session key. The AES encryption uses the CBC-128 or CBC-256 algorithm. In the source files, this section is referred to as a data block.
- The session key is RSA encrypted using the vendor's Public Key. In the source files, this section is referred to as a key block. Multiple key blocks may be present in the source file.
- The encrypted Session key and the encrypted design are merged to a file generally named the Encrypted RTL.

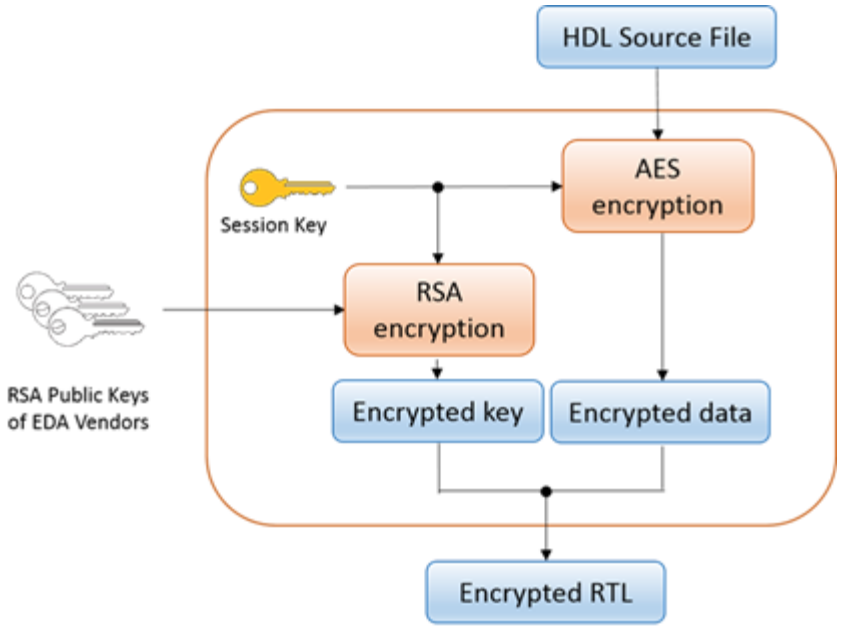
Each encrypted source file contains a single data block and one or more key blocks. The number of key blocks depends on the number of provided vendor's public keys.

Note

To decrypt an encrypted source file, you must perform the IP encryption flow steps in the reverse order.

During the next step in the design flow, typically synthesis, the Encrypted RTL is decrypted to access the original IP design, as shown in the following figure.

Figure 58: HDL Encryption Flow



By separating the encryption of data and key, you can use public keys from different vendors to encrypt the same HDL file.

For more information on how to perform HDL encryption, refer to “Running HDL Encryption from the Command Line” on page 105.

HDL File Encryption Steps

This section provides step-by-step instructions on how to encrypt an HDL file.

The Radiant software provides the key templates you can simply drag-and-drop into an HDL file. Each key template is specific to an EDA vendor providing the value of a public key.

To view the templates in Project Navigator, go to the Source Template tool. Open the **Verilog > Encryption Templates** folder and select the EDA-specific key template.

Currently, the Radiant software supports these encryption templates:

- Lattice Semiconductor
- Synplicity-1
- Synplicity-2
- Mentor
- Synopsys
- Aldec
- Cadence
- Combined Sample: provides an example of file holding multiple keys.

Step 1: Prepare the HDL file.

Annotate the HDL source file with protected pragmas. Protected pragmas provide information regarding the type of the key used to encrypt the HDL file, the name of the key, and the encryption algorithm.

In this example, the HDL source file will be encrypted by the Lattice Public Key.

```
‘pragma protect version=1
‘pragma protect encoding=(enctype=”base64”)

// optional information
‘pragma protect author=”<Your_Name>”
‘pragma protect author_info=”<Your_Information>”
```

Step 2: Specify the portion of the HDL source file to encrypt.

Annotate the HDL file to specify the encryption. Only the portion defined within these protected pragmas is encrypted.

```
‘pragma protect begin
// HDL portion that should be encrypted
‘pragma protect end
```

Step 3: Prepare key.

Define the key with which the HDL file should be encrypted. Each key definition must contain the following information:

- key_keyowner: specify the owner of the key
- key_keyname: specify the name of the key. Same owner may provide multiple keys.
- key_keymethod: specify the used cryptographic algorithm. Current version supports RSA algorithm.
- key_public_key: speficy the exact value of the key.

The key definition can be done in two ways:

Defining the key in the key.txt file: The public encryption key or keys can be defined in any .txt file. The key file may contain a single public key or a list of all available public keys. In the Radiant software, all common EDA vendor public keys are located in <*Radiant_install_directory*>/ispfpga/data/key.txt file.

The following is an example of Lattice Public Key defined in key.txt file:

```
‘pragma protect key_keyowner= "Lattice Semiconductor"
‘pragma protect key_keyname= "LSCC_RADIANT_2"
‘pragma protect key_method="rsa"
‘pragma protect key_public_key
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA0EZKUUhbuB6vSsc7OhQJ
iNAWJR5unW/OWp/LFI71eAl3s9bOYE2OlKdxbai+ndIeo8xFt2btxetUzuR6Srvh
xR2Sj9BbW1QToo2u8JfzD3X7AmRvlwKRX8708DPo4LDHZMA3qh0kfDDWkp2Eausf
LzE2cVxgq7fy/bDhUeN8xKQCSKJ7aguG6kOI6ROoZz21ljzDLUQzhm2qYF8SpU1o
tD8/uw53wLfSuhR3MBOB++xcn2imvSLqdgHWuhX6CtZIx5CD4y8inCbcLy/0Qrf6
sdTN5SAg2OZhjeNdzmqSWqhL2JTDw+Ou2fWzhEd0i/HN0y4NMr6h9fNn8nqxRyE7
IwIDAQAB
```

Defining the key directly in the HDL file: You may define the Public Key directly in the HDL file. You may define one or more keys.

```
‘pragma protect key_keyowner= "Lattice Semiconductor"
‘pragma protect key_keyname= "LSCC_RADIANT_2"
‘pragma protect key_method="rsa"
‘pragma protect key_public_key
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA0EZKUUhbuB6vSsc7OhQJ
iNAWJR5unW/OWp/LFI71eAl3s9bOYE2OlKdxbai+ndIeo8xFt2btxetUzuR6Srvh
xR2Sj9BbW1QToo2u8JfzD3X7AmRvlwKRX8708DPo4LDHZMA3qh0kfDDWkp2Eausf
LzE2cVxgq7fy/bDhUeN8xKQCSKJ7aguG6kOI6ROoZz21ljzDLUQzhm2qYF8SpU1o
tD8/uw53wLfSuhR3MBOB++xcn2imvSLqdgHWuhX6CtZIx5CD4y8inCbcLy/0Qrf6
sdTN5SAg2OZhjeNdzmqSWqhL2JTDw+Ou2fWzhEd0i/HN0y4NMr6h9fNn8nqxRyE7
IwIDAQAB
```

If the key is defined directly in the HDL file, you don't need to provide the `-k` option in the `encrypt_hdl` command.

Note

The key defined directly in an HDL source file has preference over the key defined in the key.txt file.

Step 4: Select the encryption algorithm for data encryption.

The Radiant software supports both a 128-bit and a 256-bit advanced encryption standard (AES) with CBC mode. Select the type of algorithm by defining one of the two options. The default is set to 256-bit AES with CBC mode.

```
`pragma protect data_method="aes128-cbc"
```

or

```
`pragma protect data_method="aes256-cbc"
```

Step 5: Run the encrypt_hdl Tcl command.

In the Tcl console window, type in the command to encrypt an HDL file. The option `-k` may or may not be used depending the location of the key file. The language selection (`-l`) and creation of new output file (`-o`) are optional. The default is Verilog. If you don't specify the output file, the tool generates a new output file named <*input_file_name*>_enc.v.

If the key was defined in the key.txt file: The command will encrypt the HDL file with all keys defined in the key.txt file.

```
encrypt_hdl -k <keyfile> -l <verilog | vhdl> -o <output_file>
<input_file>
```

If the key was defined directly in the HDL file:

```
encrypt_hdl -l <verilog | vhdl> -o <output_file> <input_file>
```

The encrypted file is located at the path specified in the `encrypt_hdl` command.

Step 6: Activate the encrypted HDL source file in the project file.

In the File List view, add the generated file to the project. Right-click on the encrypted file and choose **Include in Implementation**.

To see an example of a Verilog or VHDL pragma-annotated HDL source file, see "Defining Pragmas" on page 107.

Implementation Flow and Tasks

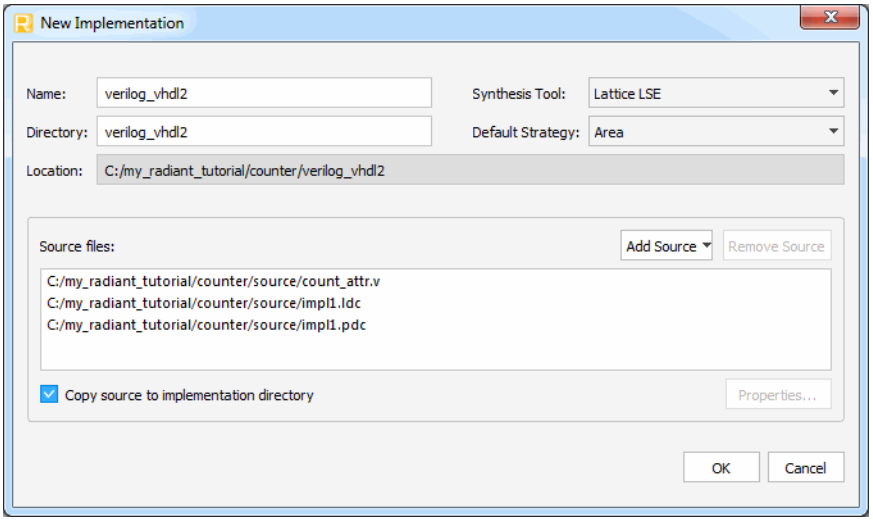
Implementations organize the structure of your design and allow you to try alternate structures and tool settings to determine which ones will give you the best results.

To help determine which scenario best meets your project goals, use a different implementation of a design using the same tool strategy, or run the same implementation with different strategies. Each implementation has an associated active strategy, and when you create a new implementation you must select its active strategy.

To try a new implementation with different strategies, you must create a new implementation/strategy combination.

1. Right-click the project name in File List.
2. Choose **Add > New Implementation**.
3. In the dialog box, choose a source file from an existing implementation using the Add Source drop-down menu.
4. Choose a strategy using the Default Strategy drop-down menu.

Figure 59: Adding a Source to a New Implementation



To use the same source for new and existing implementations, make sure that the “Copy source to implementation directory” option is not selected. This will ensure that your source is kept in sync between the two implementations.

Synthesis Constraint Creation

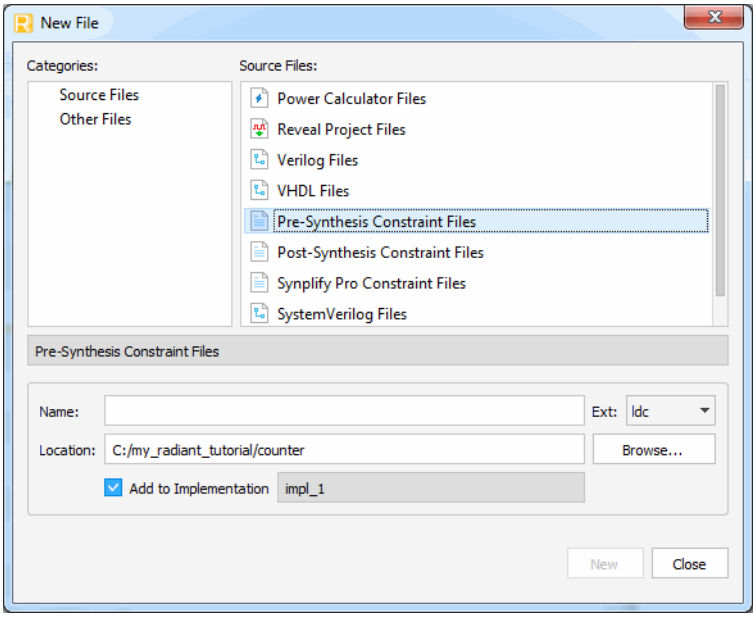
Synthesis constraints can be added to a design implementation in the format of the Synopsys Design Constraint language, while constraints can be added in the Synopsys standard timing constraints format in the form of FPGA Design Constraint (LDC, PDC, or FDC format) files.

If you are using the Lattice Synthesis Engine, the synthesis constraints will be included in an .ldc file. If you are using Synplify Pro for synthesis, the constraints will be included in an .fdc file. The older .sdc file format is also supported for constraints.

To create a new synthesis constraint file, right-click the Synthesis Constraint Files folder in the File List pane and choose **Add > New File**. In the New File dialog box, select one of the following and give the file a name:

- Pre-Synthesis Constraint Files (.ldc)
- Post-Synthesis Constraint Files (.pdc)
- Synplify Pro Constraint Files (.fdc)

Figure 60: New Synthesis Constraint Files



The .ldc, .pdc, or .fdc file will open in the Source Editor to allow you to manually add the constraints. You can use the Pre-Synthesis Timing Constraint Editor tool to add pre-synthesis timing constraints to the .ldc file and the Post-Synthesis Timing Constraint Editor tool to add logic view level post-synthesis timing constraints to .pdc files. You can also use the Device Constraint Editor and Floorplan View to add physical constraints to .pdc files. For detailed information about setting constraints, see *Applying Design Constraints* and the *Constraints Reference Guide* in the Radiant online Help.

An alternative way of adding constraint files is through Source Template. To view a constraint template, click on the Source Template tab on the left-hand side of the Project Navigator pane. If not selected, make sure it is enabled in **View > Show Views > Source Template**. The list of constraint templates includes the timing constraints, physical constraints, and user templates. Select a template and copy and paste it into your active design.

Constraint Creation

LDC (pre-synthesis) and PDC (post-synthesis) files are used to input timing and physical constraints. The following steps illustrate how to assign and edit constraints in the Radiant software and implement them at each stage of the design flow.

1. If desired, define some constraints at the HDL level using HDL attributes. These source file attributes are included in the Unified Database (UDB), and will be displayed in the Radiant software after the Map Design process is run. The following is an example of applying the LOC attribute in Verilog source code:

   ```
   module top (
     input clk1,
     input datain /* synthesis loc = B12 */,
     output ff_clk1out
   )
   ```

   For more information on HDL Attributes, see the topic “HDL Attributes” in the *Constraints Reference Guide* section of the Radiant online Help.

2. Open one or more of the following tools to create new constraints or to modify existing constraints from the source files:
   - Device Constraint Editor, which consists of:
     - Spreadsheet View – the primary view for setting constraints.
     - Package View – examines the pin layout of the design, modifies signal assignments, reserves pin sites that should be excluded from placement and routing, and runs PIO design rule check to verify legal placement of signals to pins.
     - Device View – examines FPGA device resources.
     - Netlist View - shows port types (Input, Output) and groups.
   - Timing Constraint Editor. Timing/Physical constraints are entered through:
     - Pre-Synthesis Timing Constraint Editor - used to enter pre-synthesis timing constraints such as clocks, clock latency/ uncertainty/Group, Input/Output delays, timing exceptions, and attributes.
     - Post-Synthesis Timing Constraint Editor - This post-synthesis version of the Timing Constraint Editor is used to enter logic view level timing constraints and physical constraints.
   - Floorplan View examines the device layout of the design, draws bounding boxes for GROUPs, draws REGIONs for the assignment of groups or to reserve an area, and reserves sites and REGIONs that should be excluded from placement.
3. Save the constraints to the pre-synthesis constraint file (.ldc) or post-synthesis constraint file (.pdc).
4. Run the Map Design process (Map).
5. Run the Map Timing Analysis process and examine the timing analysis report. This is an optional step, but it can be a quick and useful way to

identify serious timing issues in the design and constraint errors (syntax and semantic). Modify constraints as needed and save them.

6. Run the Place & Route Design process.
7. Open views directly or by cross-probing to examine timing and placement and create new GROUPs. Also examine the Place & Route Timing report.
8. Modify constraints or create new ones using the appropriate constraint tool. Save the constraint changes and rerun the Place & Route Design process.

Simulation Flow

The simulation flow in the Radiant software supports source files that can be set in the File List view to be used for the following purposes:

- Simulation and Synthesis (default)
- Simulation
- Synthesis

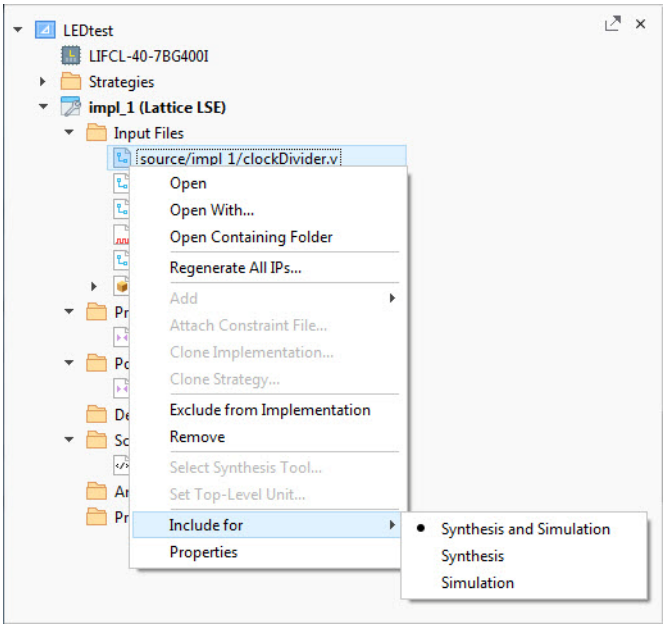
This allows the use of test benches, including multiple file test benches. Additionally, multiple representations of the same module can be supported, such as one for simulation only and one for synthesis only.

You can add top level signals to the waveform display in the simulator and to automatically start the simulator running.

The Simulation Wizard automatically includes any files that have been set for simulation only or for both simulation and synthesis. You can select the top of the design for simulation independent of the implementation design top. This allows easy support for test bench files, which are normally at the top of the design for simulation but not included for implementation. The implementation wizard exports the design top to the simulator, along with source files, and set the correct top for the .spf file if running timing simulation.

After you add a module, use the **Include for** menu to specify how the module file is to be used in the design.

Figure 61: "Include For" Input Files



Simulation Wizard Flow

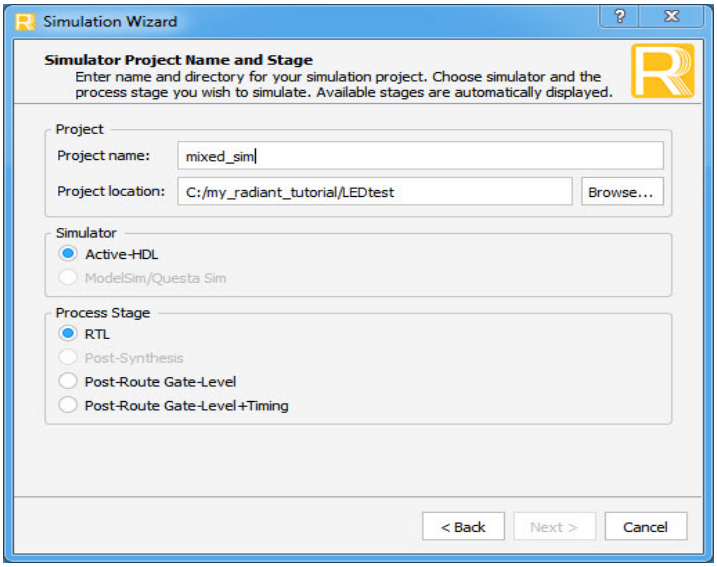
When you are ready to simulate, export the design using the Simulation Wizard. Aside from the RTL simulation, you can perform the Post-Synthesis simulation and Post-Place & Route back annotated netlist simulation.

Post-Synthesis Simulation File generates a Verilog netlist (_syn.vo) file. Similarly, the Gate-Level Simulation File generates a Verilog netlist of the routed design (.vo file) that is back annotated with timing information (.sdf file). The generated file enables you to run a timing simulation of your design. For more details on how to generate these files, see "Task Detail View" on page 32.

Choose **Tools > Simulation Wizard** or click the Simulation Wizard icon on the toolbar. The wizard leads you through a series of steps:

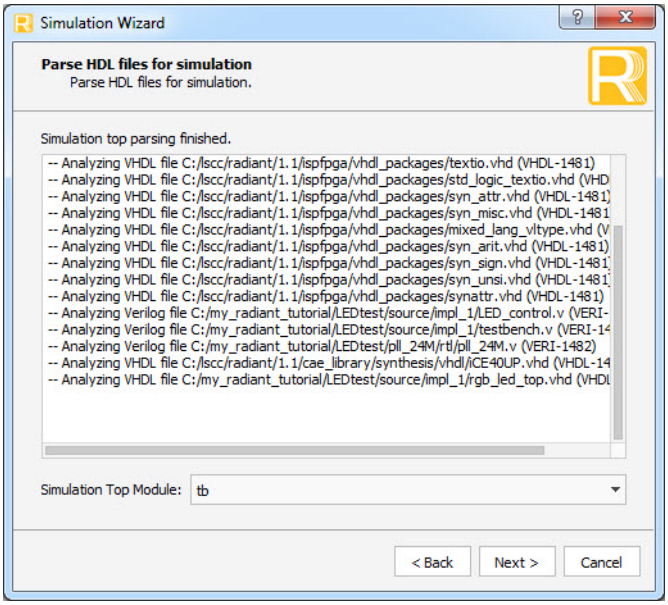
1. Select a simulation project name and location.
2. Specify the simulator to use (if you have more than one installed).
3. Select the process stage to use (RTL, Post-Synthesis, or Post-Route Gate-Level + Timing).
4. Select the source files.

Figure 62: New Simulation Project



After you have set up the simulator project and specified the implementation stage and source files to be included, the Simulation Wizard parses the HDL and test bench. The last step is to specify the simulation top module.

Figure 63: Simulation Top Module



In some designs, the compile order of the HDL files passed to the simulator might result in compilation warnings. In most cases, these compilation warnings can be safely ignored. The warnings can be eliminated in one of two ways:

- The correct compilation order for the HDL files can be set manually in the File List view. After the correct order is determined, the files are sent to the simulator, which will eliminate any compilation warnings.

- The correct compilation order for the HDL files can be set in the Simulation Wizard during the “Add and Reorder” step. After the correct order for the files is set manually, the files will be sent to the simulator, which will eliminate any compilation warnings.

You can also run the simulation directly from the wizard.

Chapter 7

Working with Tools and Views

This chapter covers the tools and views controlled from the Radiant software framework. Tool descriptions are included and common tasks are described.

Overview

The Radiant software design environment streamlines the implementation process for FPGAs by combining the tool control and data views into one common location. Two main features of this design environment make it easy to keep track of unsaved changes in your design and examine data objects in different view.

Shared Memory

The Radiant software uses shared memory that is accessed by all tools and views. As soon as design data has been changed, an asterisk * appears in the tab title of the open views, notifying you that unsaved changes are in memory.

Cross-Probing

Shared design data in the Radiant software enables you to select a data object in one view and display it in another. This cross-probing capability is especially useful for displaying the physical location of a component or net after it has been implemented. You can click on a hyperlink icon to cross-probe into the specific tool. The Radiant software supports:

- Post-Synthesis timing report links to Netlist Analyzer
- Map & PAR timing reports link to Physical Designer's Placement and Routing views

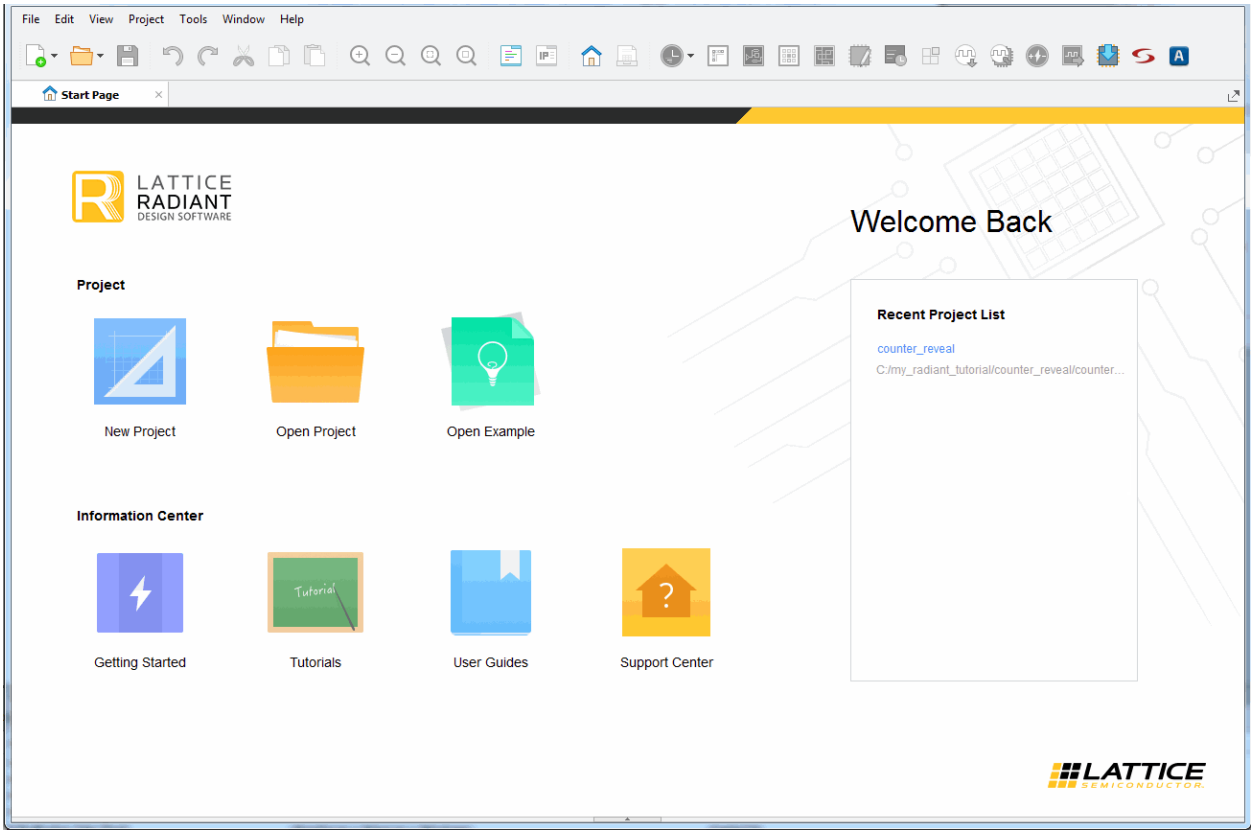
View Menu Highlights

The View menu and toolbar control the display of all toolbars, project views and display control. Also included in the View menu are the important project-level features: Start Page, and Reports.

Start Page

The Start Page is displayed by default when you run the Radiant software. The Start Page enables you to open projects, read product documentation, and view the software version and updates. You can modify startup behavior by choosing **Tools > Options > General > Startup**.

Figure 64: Default Start Page

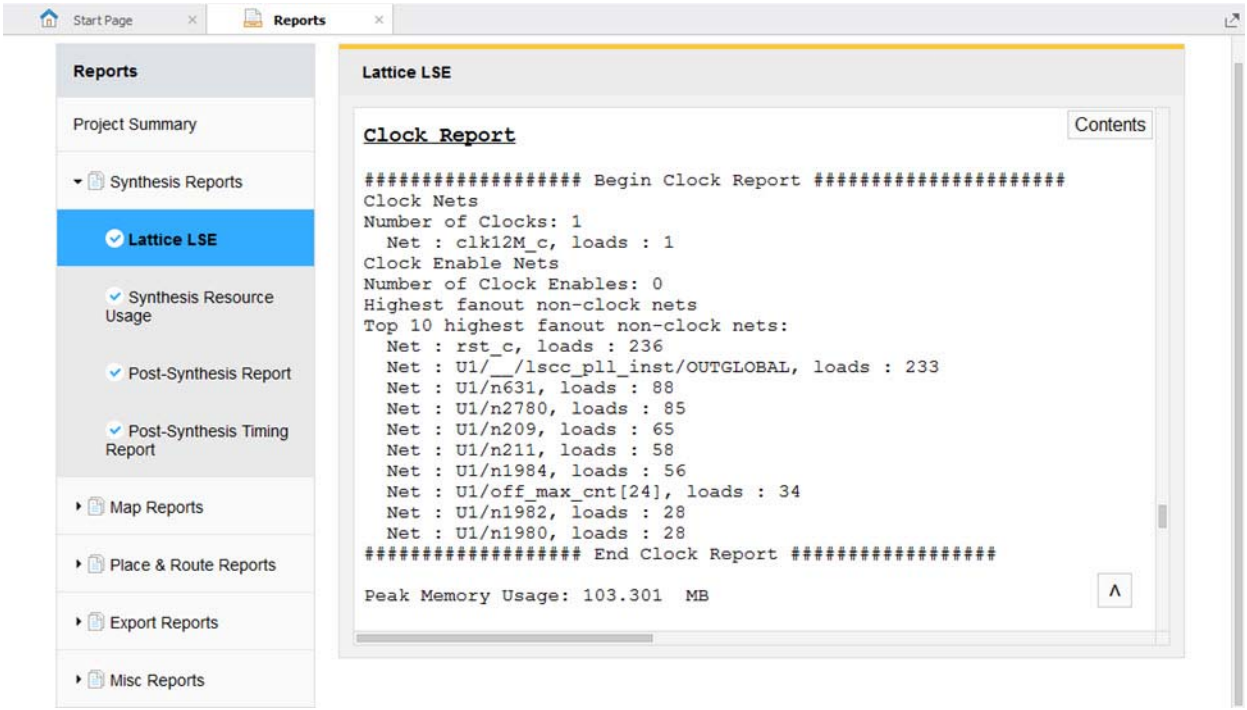


The Start Page gives you quick access to recent projects and to product documentation.

Reports

The Reports view provides one central location for the project summary and design processing reports. It is displayed by default when a project is open. Alternatively, click on the Reports icon in the toolbar.

Figure 65: Reports View Showing Last Process Run



The Reports view is organized into Project Summary, Synthesis Reports, Map Reports, Place & Route Reports, Export Reports, and Misc Reports.

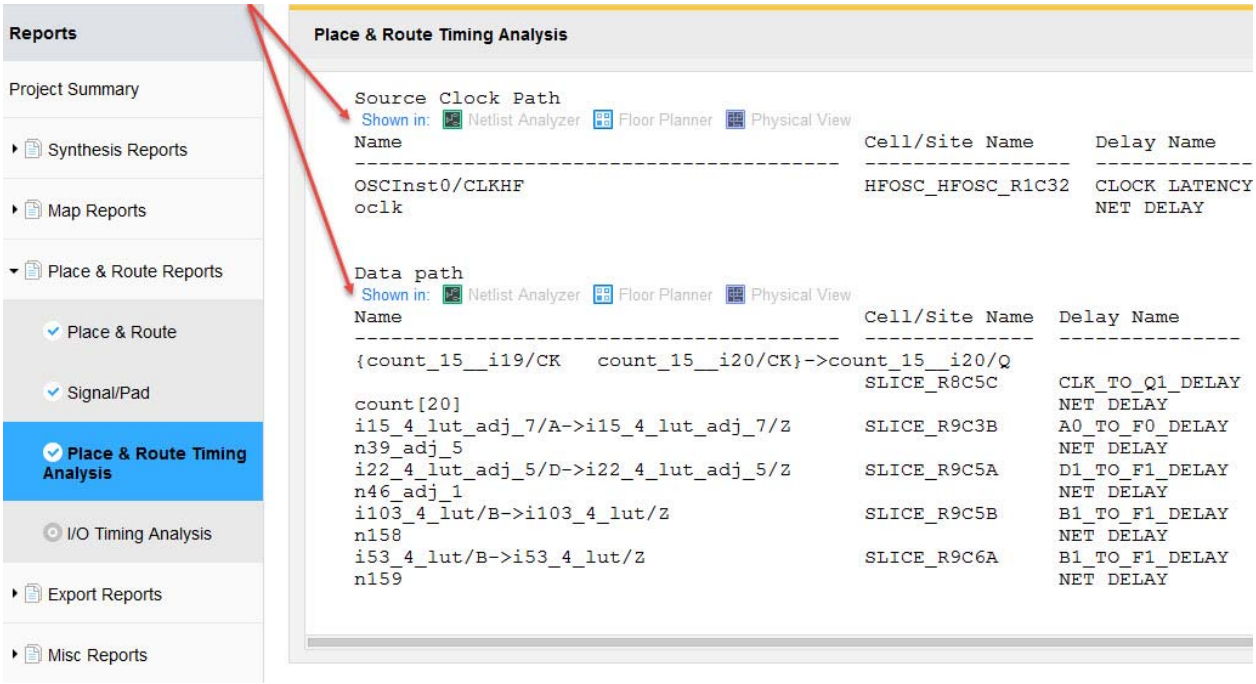
The different file icons indicate:

- A report has been completed (blue check mark).
- A report has never been generated (gray circle).
- A report is out of date (orange question mark).

Select any item to view its report.

The Reports view also supports path cross-probing through the timing or analysis reports. You can view a specific clock or data path in Netlist Analyzer and Physical Designer by clicking the icon next to the path.

Figure 66: Cross-Probing



To learn more about cross-probing, view “Cross-Probing” on page 38.

Tools

The entire FPGA implementation process tool set is contained in the Radiant software. You can run a tool by selecting it from the Tools menu or toolbar.

This section provides an overview of each of these tools. More detailed information is available in their respective user guides, which you can access from the Start Page or from the Radiant software Help. Detailed descriptions of external tools can be found in their product documentation as well.

If you are viewing an encrypted design, some secured objects may not be visible in the selected tool. To learn more, see “Secure Objects in the Design” in the online help.

Timing Constraint Editor

The Timing Constraint Editor is used to edit pre-synthesis timing (.ldc) constraints and post-synthesis constraints stored in a .pdc file.The Timing Constraint Editor consists of two tools:

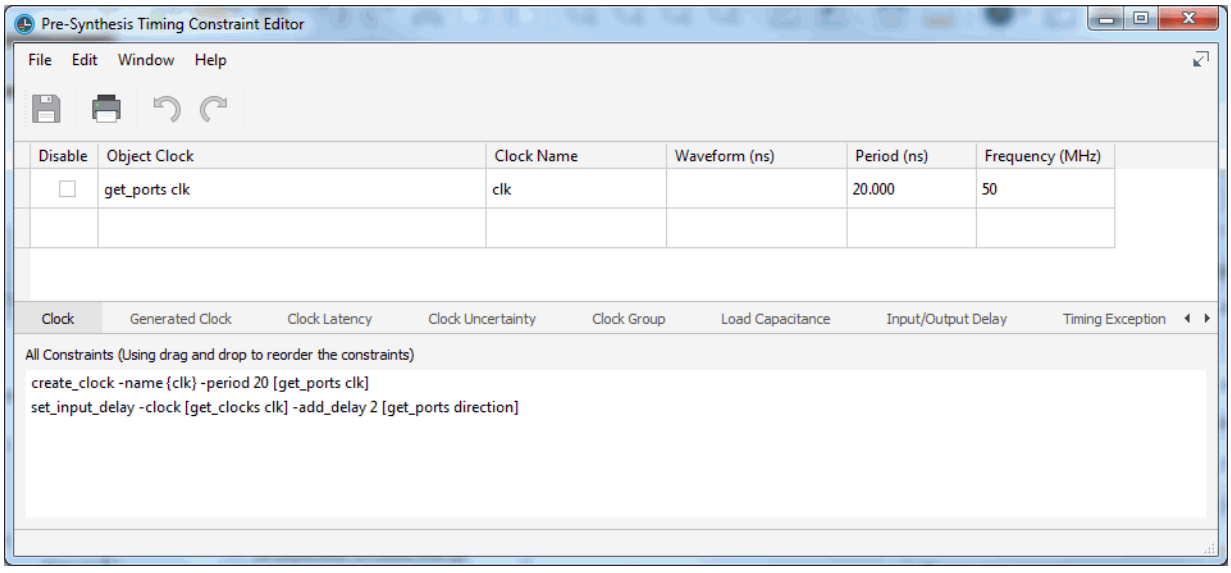
- Pre-Synthesis Timing Constraint Editor
- Post-Synthesis Timing Constraint Editor

Both tools have identical interfaces and the entry mechanisms of the constraints are also the same. The key differences are that pre-synthesis constraints are entered pre-synthesis and are synthesized by the chosen synthesis tool. The post-synthesis constraints are already synthesized and populated in each of the different constraints tabs. You cannot modify the post-synthesis constraints that were populated from pre-synthesis, but can supply new constraints (physical and timing) to either override the existing one already populated or supply new constraints to better constrain the design for improved performance upon analysis.

The different constraint types are entered through these tabs:

- Clock - used to define the clocking scheme of the design.
- Generated Clock - used to define generated clocks such as from PLLs.
- Clock Latency - is the delay between the clock source and clock pin. Used to define the latency in terms of rise and fall times.
- Clock Uncertainty - is the jitter difference of two signals, possibly caused by clocks. Used to define the amount of uncertainty of a clock or during clock domain transfer.
- Clock Group - used to specify which clocks are not related in terms of being logically/physically exclusive and asynchronous.
- Load Capacitance - used to define the load capacitance of ports.
- Input/Output Delay - used for setting input and output delays.
- Timing Exception - used to set min/max, false, and multicycle path constraints.
- Attribute - used to set synthesis attributes.

Figure 67: Pre-Synthesis Timing Constraint Editor



Constraint Propagation

Constraint propagation is not a standalone tool, nor do you have to provide any input to take advantage of this feature.

You usually define constraints for the top level design as well as include other constraint files for any IP that are generated with Radiant tools such as IP Catalog. Constraints defined at the module IP level may not contain the correct hierarchical names and hence will not be applied correctly when synthesized. To help honor as much of the supplied constraints as possible, this feature runs on all the input constraints a DRC that writes out a new constraint file to support hierarchical constraints such as soft-IP constraint and honor top level constraints.

For more information about Constraint Propagation, refer to *Constraint Propagation* in the Radiant software online Help.

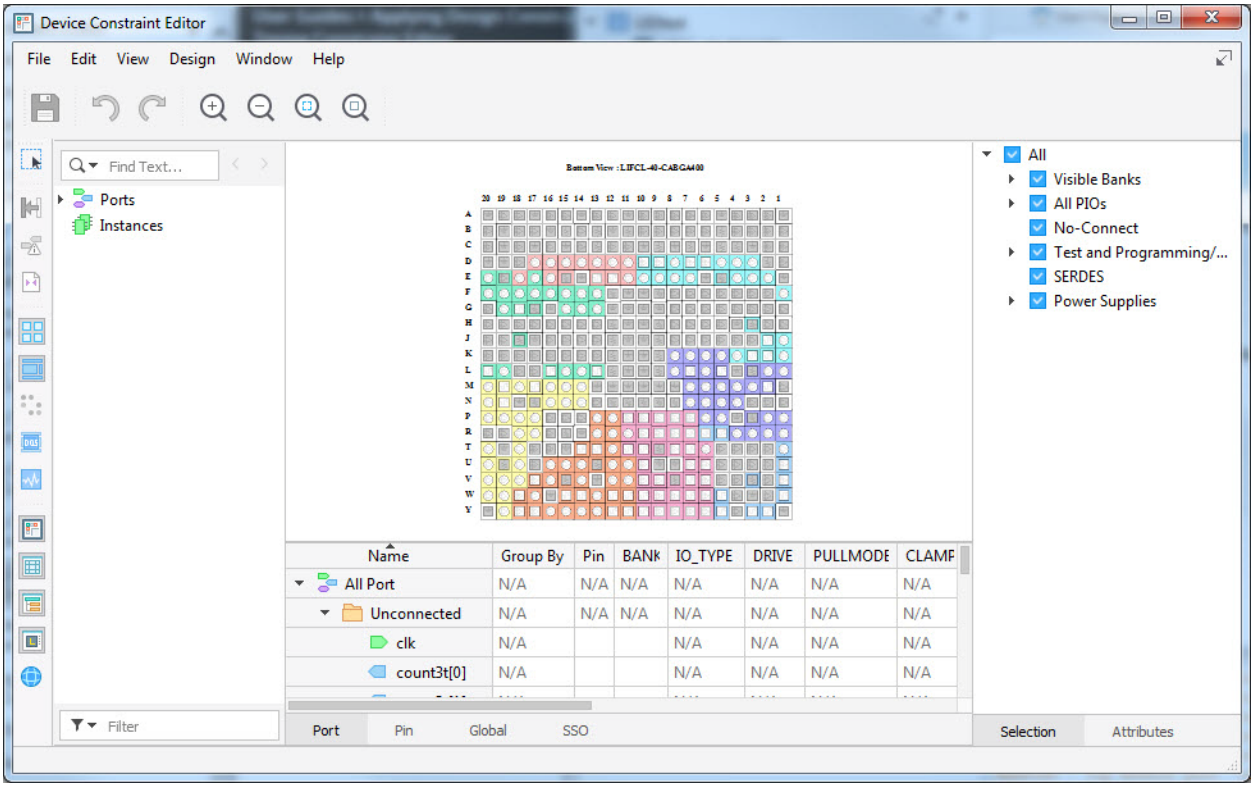
Device Constraint Editor

The Device Constraint Editor is used to edit post-synthesis constraints. These constraint editing views are available from the Radiant toolbar and Tools menu.

All modified constraints are saved to a .pdc file and the flow returned to Map. The Device Constraint Editor shows the pin layout of the device and displays the assignments of signals to device pins. This view allows you to edit these assignments, and reserve sites on the layout to exclude from placement and routing. The Device Constraint Editor is also the entry mechanism for physical constraints.

Device Constraint Editor views, shown in Figure 68, enable you to develop constraints that will shorten turn-around time and achieve a design that conforms to critical circuit performance requirements.

Figure 68: Device Constraint Editor



Global Settings

In the Device Constraint Editor tool at the bottom of the tool, there are a series of tabs to allow you to set many of the device settings such as Junction Temperature, Voltage, SysConfig, User Code, Derating, Bank VCCIO, Global Set/Reset, Use Primary Net, and Vref Locate constraints.

For more information on how to do this, in addition to detailed information about Device Constraint Editor, see *Setting Global Settings* in the Radiant online Help.

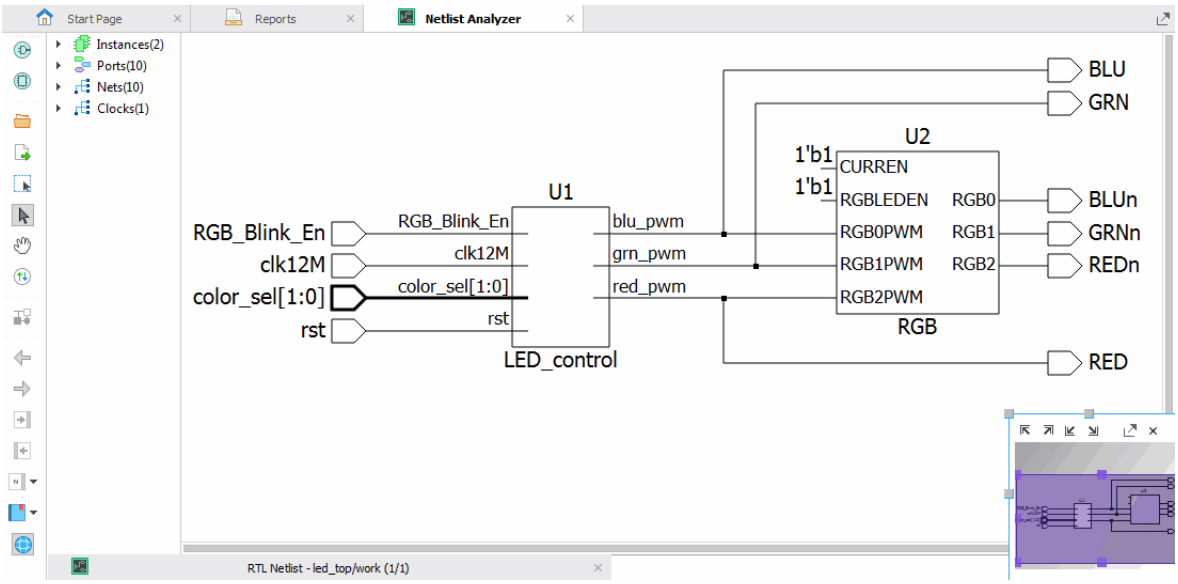
SSO Analysis

Radiant enables you to run an analysis of simultaneous switching outputs (SSO). SSO analysis describes the noise on signals caused by a large number of out drivers that are switching at the same time. Analysis of SSO helps ensure that your I/O standards and power integrity requirements of the PCB design. This tool can be accessed in the Device Constraint Editor in the SSO tab at the bottom. From there a user can set output loads, Ground plane PCB noise, SSO Allowance %, and Power plane PCB noise values. Once the flow is re-run, an SSO report is generated.

Netlist Analyzer

Netlist Analyzer works with Lattice Synthesis Engine (LSE) to produce schematic views of your design while it is being implemented. (Synplify Pro also provides schematic views.) Use the schematic views to better understand the hierarchy of the design and how the design is being implemented.The Netlist Analyzer window has four parts, as shown in Figure 69.

Figure 69: Netlist Analyzer



- Tool bar provides buttons for various functions.
- Browser provides nested lists of module instances, ports, and nets.
- Schematic view shows a schematic of the design. Depending on the size of the design, the schematic may be made of multiple sheets.
- World View, which is a miniature view of the sheet, helps you pan and zoom in the schematic view.

Netlist Analyzer can have multiple schematics open. The open schematics are shown on tabs along the bottom of the window.

There are several ways to adjust the view of a schematic and to navigate through the hierarchy. For more information on how to do this, in addition to detailed information about Netlist Analyzer, see *About Netlist Analyzer* in the Radiant online Help.

Physical Designer

The Physical Designer is a combined GUI interface of both the Placement Mode and Routing Mode accessible within this one Radiant software tool. This provides for one central location where you can do all the floorplanning and be able to view the physical layout of the design.

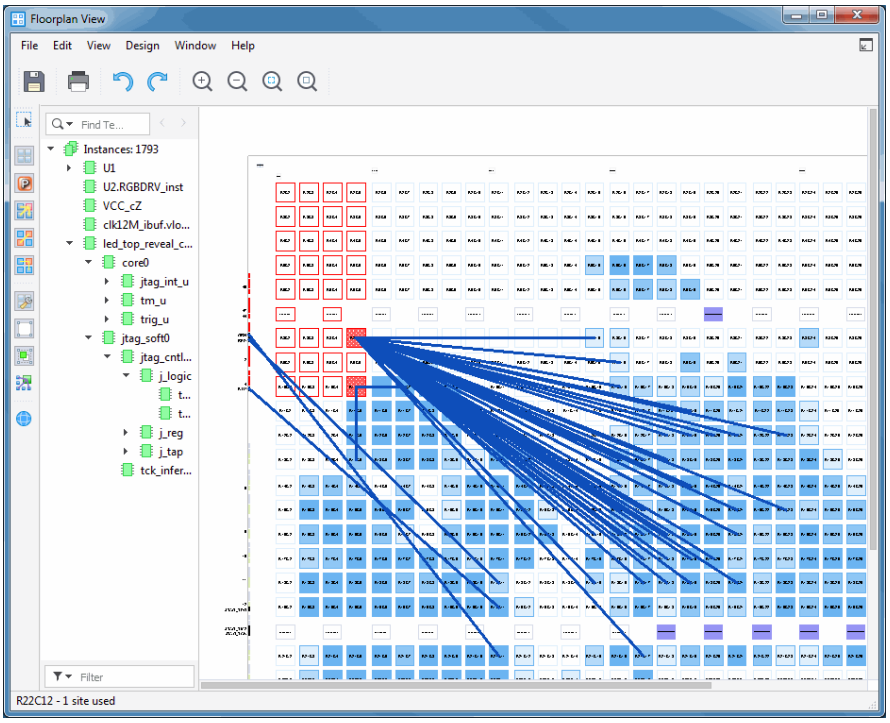
Placement Mode

Placement Mode provides a large-component layout of your design, and is available as soon as the target device has been specified. Placement Mode displays user constraints, and placement and routing information. All connections are displayed as fly-lines.

Placement Mode allows you to create REGIONs and bounding boxes for GROUPs, in addition to specifying the types of components and connections to be displayed. As you move your mouse pointer over the floorplan layout, details are displayed in tool-tips and in the status bar, including:

- Number of resources for each GROUP and REGION
- Number of utilized slices for each PLC component
- Name and location of each component, port, net, and site

Figure 70: Placement Mode



Placement Mode has a Congestion Timing Map view allowing you to debug timing congested areas of your design. This view gives the timing of the most critical paths within the slack threshold input. Furthermore, the Congestion View is a read-only view that shows the most congested regions based on wire length and pins selected.

Figure 71: Congestion Timing Map View

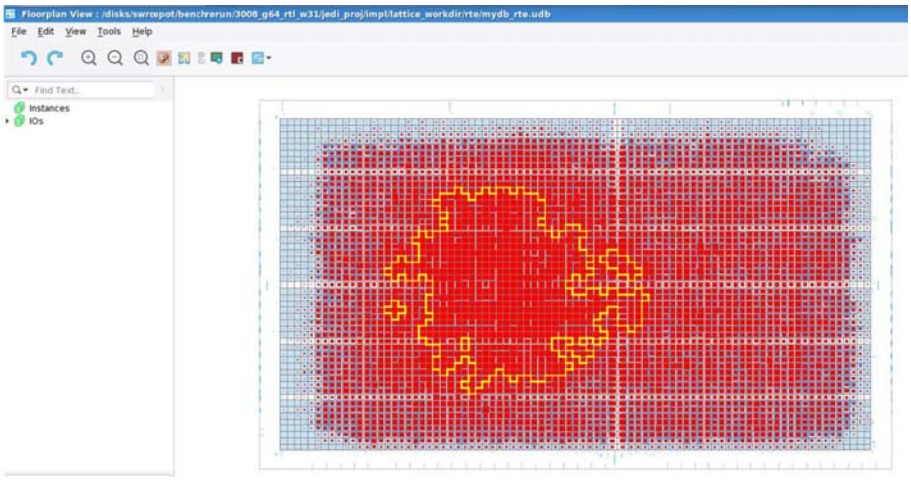
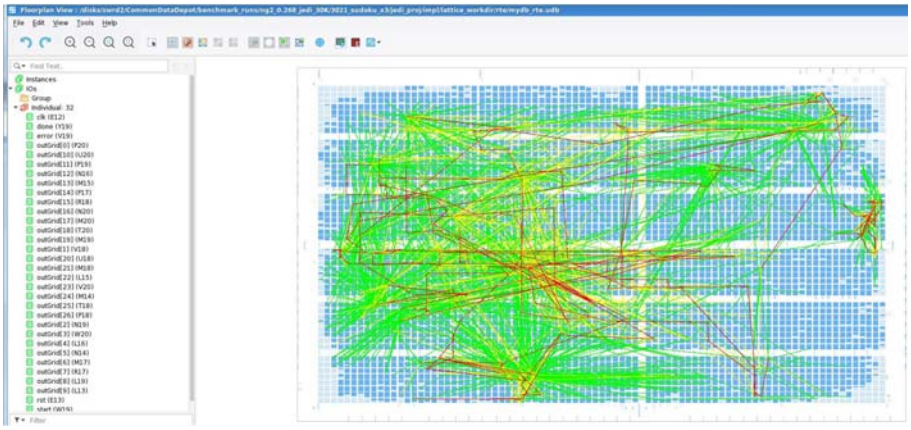


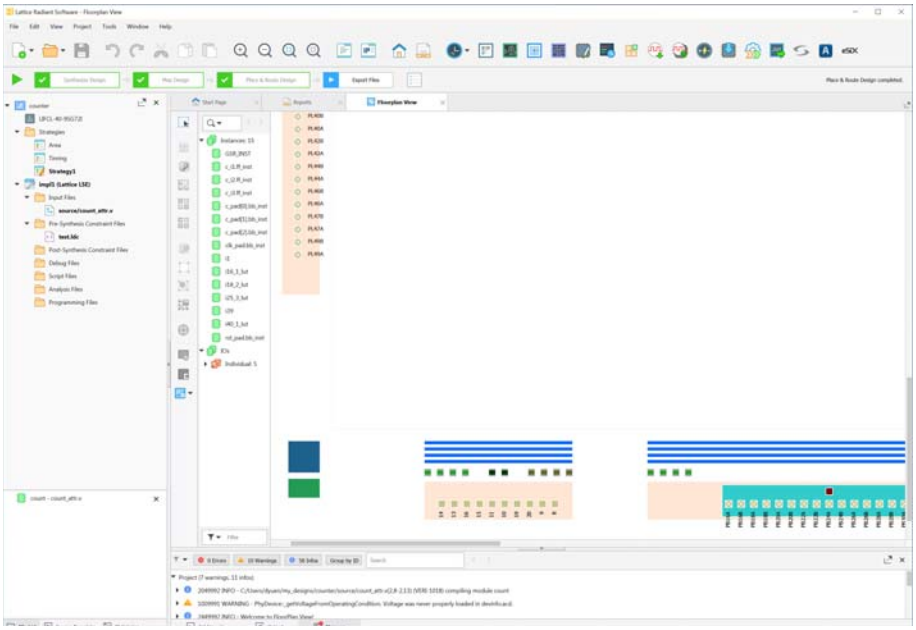
Figure 72: Congestion View



IO Planner View

Stemming from the Floorplan view is an IO Planner view. Floorplan view is for GROUPS and REGIONS assignment. IO Planner is used for IO assignment such as DDR Interface, DQS and clock assignments. IO resource utilization can be displayed by hovering over and displayed in tool tips.

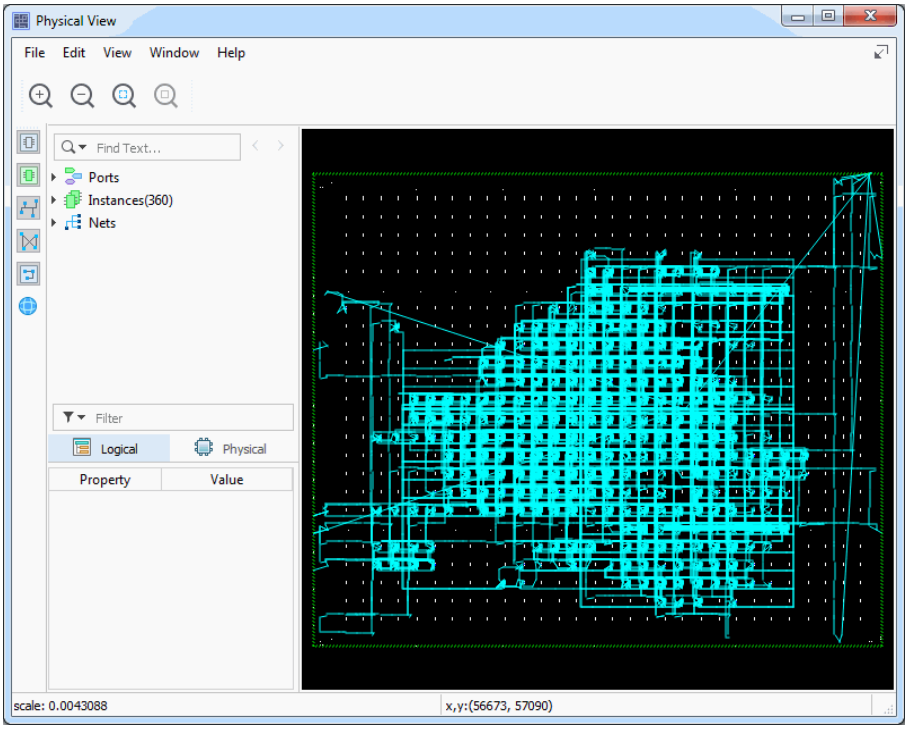
Figure 73: IO Planner View



Routing Mode

Routing Mode provides a read-only detailed layout of your design that includes physical wire connections. Routed connections are displayed as Manhattan-style lines, and unrouted connections are displayed as fly-lines.

Figure 74: Routing Mode



As you move your mouse pointer slowly over the layout, the name and location of each REGION, group, component, port, net, and site are displayed as tool tips and also appear in the status bar. The tool tips and status bar also display the group name for components that are members of a group.

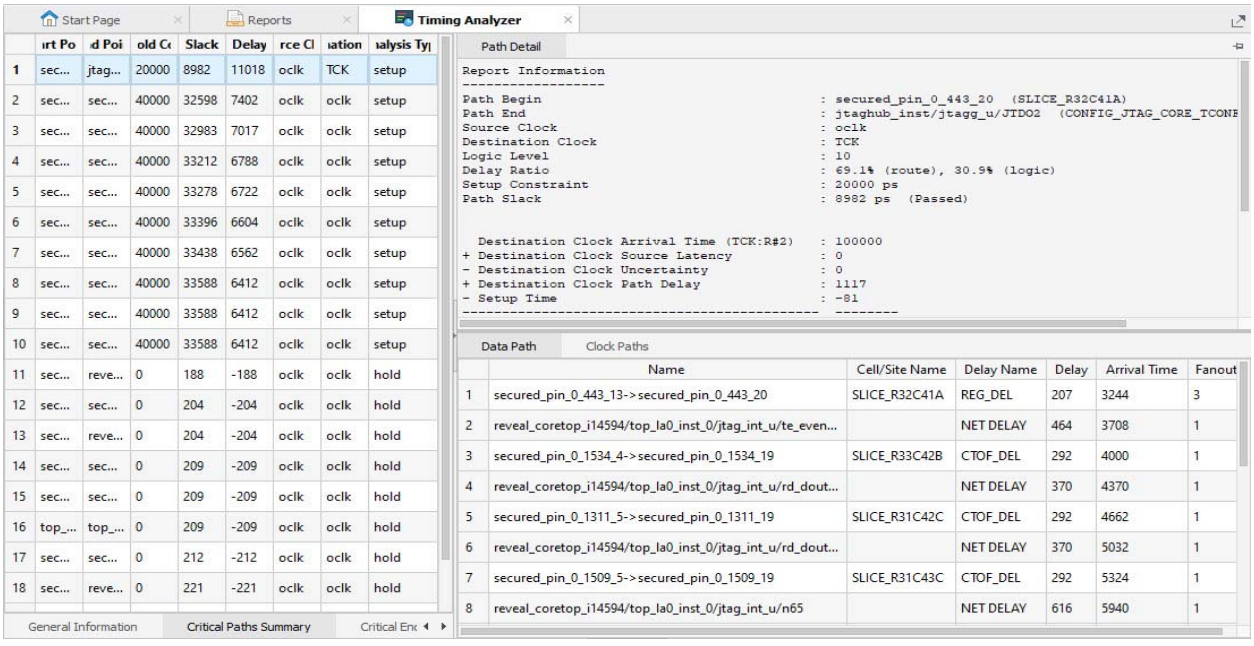
The Routing Mode toolbar allows you to select the types of elements that will be displayed on the layout, including components, empty sites, pin wires, routes, and timing paths. Routing Mode is available after placement and routing.

Timing Analyzer

Timing Analyzer analyzes timing constraints that are present in the .ldc and .pdc files. These timing constraints are defined in the Timing Constraint Editor or in a text editor before the design is mapped. A Timing Analysis report file, which shows the results of timing constraints, is generated each time you run the LSE, Map Timing Analysis process, or the Place & Route Timing Analysis (PAR) process. Place & Routing Timing Analysis results can then be viewed in the Timing Analyzer windows.

The Map Timing Analysis report (.tw1) contains estimated routing that can be used to verify the expected paths and to provide an estimate of the delays before you run Place & Route. The PAR Timing Analysis report (.twr) contains delays based on the actual placement and routing and is a more realistic estimate of the actual timing.

Figure 75: Timing Analyzer



Timing Analyzer consists of five tabs of information:

- General Information

- Critical Paths Summary
- Critical Endpoint Summary
- Unconstrained Endpoint Summary
- Query

Each tab can be detached from the main window, rearranged, and resized. When you select a constraint from the Constraint pane, you can view the path table details on one pane, and Timing Analyzer report in the other. For detailed information about Timing Analyzer see *Analyzing Static Timing* in the Radiant online Help.

Reveal Inserter

Reveal Inserter lets you add debug information to your design to allow hardware debugging using Reveal Analyzer. Reveal Inserter enables you to select the design signals to use for tracing, triggering, and clocking. Reveal Inserter will automatically generate the debug core, and insert it into a modified design with the necessary debug connections and signals. Reveal Inserter supports VHDL and Verilog sources. After the design has been modified for debug, it is mapped, placed and routed with the normal design flow in the Radiant software.

For more information about Reveal Inserter, refer to the *Reveal User Guide for Radiant Software*. Also, refer to T*esting and Debugging On-Chip* in the Radiant online Help.

Reveal Analyzer

After you generate the bitstream, you can use Reveal Analyzer to debug your FPGA circuitry. Reveal Analyzer gives you access to internal nodes inside the device so that you can observe their behavior. It enables you to set and change various values and combinations of trigger signals. After the specified trigger condition is reached, the data values of the trace signals are saved in the trace buffer. After the data is captured, it is transferred from the FPGA through the JTAG ports to the PC.

For more information about Reveal Analyzer, refer to the *Reveal User Guide for Radiant Software*. Also, refer to *Testing and Debugging On-Chip* in the Radiant online Help.

Reveal Controller

Reveal Controller allows you to emulate an otherwise unavailable environment for power debug. For example, your evaluation board would only have a limited number of LEDs or switches but the virtual environment enables up to 32 bits. Register memory mapping and dumping of values is also easily manifested while visibility into Hard IP is also enabled.

For more information about Reveal Controller, refer to the *Reveal User Guide for Radiant Software*. Also, refer to *Testing and Debugging On-Chip* in the Radiant online Help.

Power Calculator

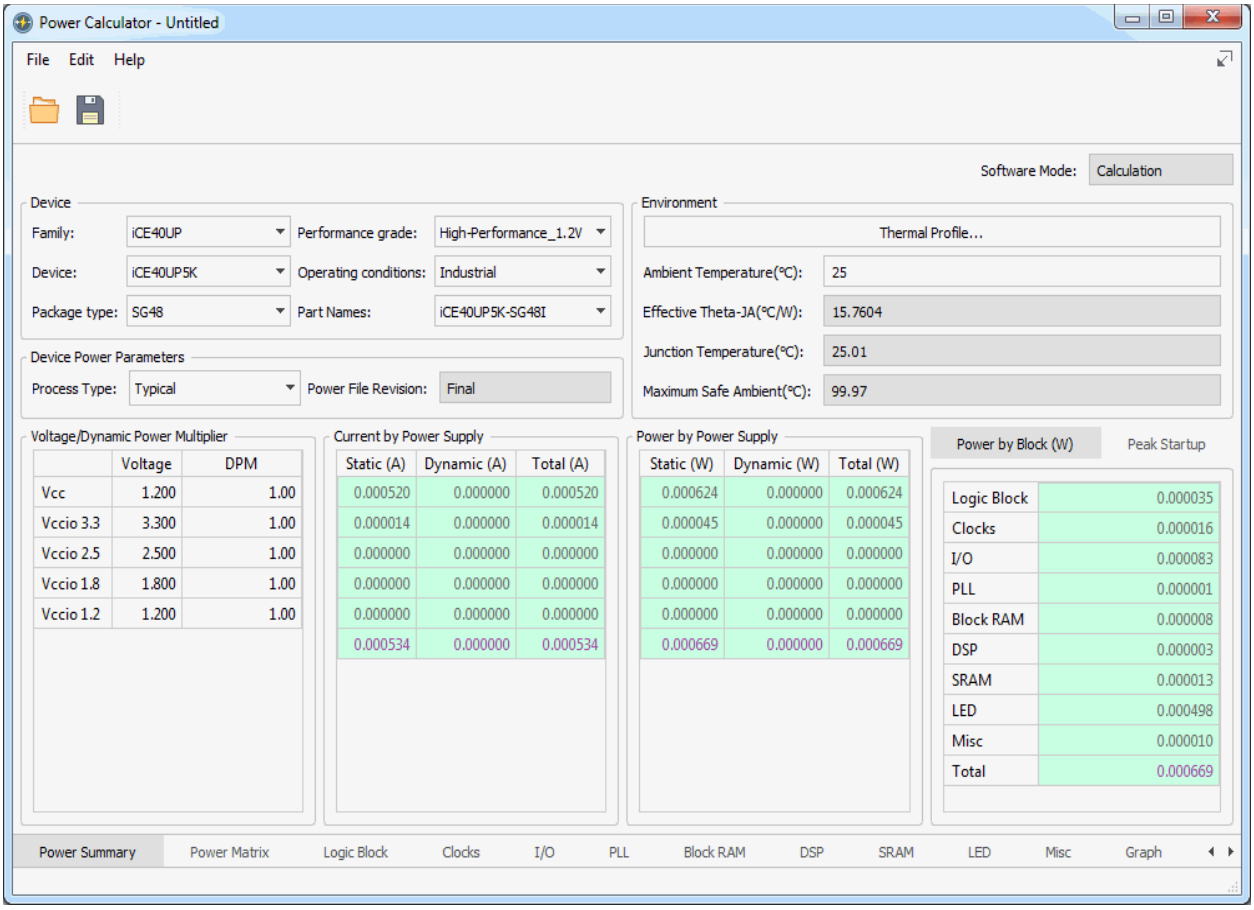
Power Calculator estimates the power dissipation for your design. It uses parameters such as voltage, temperature, process variations, air flow, heat sink, resource utilization, activity and frequency to calculate the device power consumption. It reports both static and dynamic power consumption.

To launch Power Calculator from the Radiant software, choose **Tools > Power Calculator** or click the Power Calculator button on the toolbar.

Power Calculation Modes Power Calculator opens in estimation mode or calculation mode, depending on the status of the selected .pcf file. If it opens in calculation mode, the Bank settings will be from background power database, not from the constraint file. When you make certain design changes in calculation mode, Power Calculator reverts to estimation mode.

Power Calculator Pages When Power Calculator opens, it displays the Power Summary page, which enables you to change the targeted device, operating conditions, voltage, and other basic parameters. Updated estimates of power consumption are then displayed based on these changes. Tabs for other pages, including Power Matrix, Logic Block, Clocks, I/O, I/O Term, Block RAM, Graph, and Report, are arranged across the bottom. The number and types of these pages depends on the target device.

Figure 76: Power Calculator



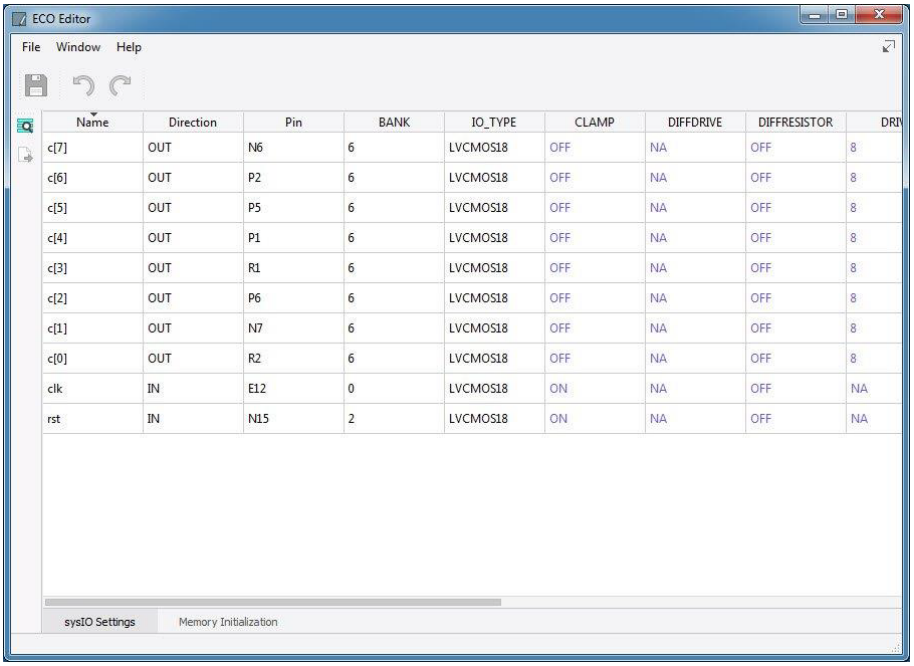
Power Calculator is also available as a non-integrated tool, which you can launch without opening the Radiant software. The non-integrated Power Calculator provides all the same functionality as the integrated version. To open the non-integrated Power Calculator from the Windows Start menu, select **Lattice Radiant Software > Accessories > Power Calculator**. The Startup Wizard enables you to create a new Power Calculator project, based on a selected device or a processed design, or to open an existing Power Calculator project file (.pcf).

For more information on Power Calculator see *Analyzing Power Consumption* in the Radiant online Help.

ECO Editor

The Engineering Change Order (ECO) Editor enables you to safely make changes to an implemented design without having to rerun the entire process flow. Choose **Tools > ECO Editor** or click the ECO Editor button on the toolbar.

Figure 77: ECO Editor



ECOs are requests for small changes to be made to your design after it has been placed and routed. The changes are directly written into the Unified Database (.udb) file without requiring that you go through the entire design implementation process.

ECOs are usually intended to correct errors found in the hardware model during debugging. They are also used to facilitate changes that had to be made to the design specification because of problems encountered when other FPGAs or components of the PC board design were integrated.

The ECO Editor includes windows for editing I/O preferences, and memory initialization values. It also provides a Change Log window that enables you to track the changes between the modified .udb file and the post-PAR .udb file.

Note

After you edit your post-PAR UDB file, your functional simulation and timing simulation will no longer match.

For more information, see *Applying Engineering Change Orders* in the Lattice Radiant online Help.

Programmer

The Radiant Programmer is a system for programming devices. The software supports serial programming of Lattice devices using PC and Linux environments. The tool also supports embedded microprocessor programming.

For more information about Programmer and related tools, refer to the *Programming Tools User Guide for Radiant Software*. Also, refer to *Programming the FPGA* in the Radiant software or Stand-Alone Programmer online Help.

Run Manager

Run Manager runs the processes for the different implementation/strategy combinations. Choose **Tools > Run Manager** or click the Run Manager button on the toolbar.

Run Manager takes the design through the entire process flow for each selected implementation. If you are running on a multi-core system, Run Manager will distribute the iterations so that they are executed in parallel. The option “Maximum implementation processes running in Run Manager” is available in the General section of the Options dialog box. Choose **Tools > Options** to access it. This option enables you to set the maximum number of processes to run in parallel. Generally, the maximum number of processes should be the same as the number of cores in your processor; but if the strategy is using the “Multi-Tasking Node List” option for Place & Route Design, this number should be set to one.

You can use the Run Manager list to set an implementation as active. Right-click the implementation/strategy pair and choose **Set as Active**.

For an implementation that uses multiple iterations of place-and-route, you can select the iteration that you want to use as the active netlist for further processes. Expand the implementation list, right-click the desired iteration, and choose **Set as Active**. The active iteration is displayed in italics.

To examine the reports from each process, set an implementation as active, and then select the Reports view.

See the *Managing Projects* section of the Radiant online Help for more information about using implementations, strategies, and Run Manager.

Synplify Pro for Lattice

Synplify Pro for Lattice is an OEM synthesis tool used in the Radiant software design flow. Synplify Pro runs in batch mode when you run the Synthesize Design step in Process View. To examine the output report, select **Synplify Pro** in the Process Reports folder of Reports View.

You can also run Synplify Pro in interactive mode. Choose **Tools > Synplify Pro for Lattice** or click the Synplify Pro button on the toolbar.

For more information, see the *Synplify Pro User Guide*, which is available from the Radiant Start Page or the Synplify Pro Help menu.

Active-HDL Lattice Edition

The Active-HDL Lattice Edition tool is an OEM simulation tool that is closely linked to the Radiant software environment. It is not run as part of the Process implementation flow. To run Active-HDL, choose **Tools > Active-HDL Lattice Edition** or click the Active-HDL button on the toolbar.

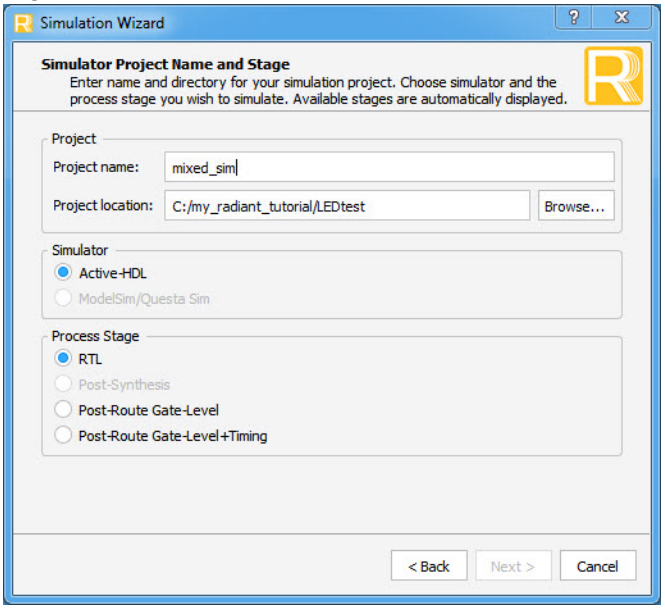
See "Simulation Flow" on page 67 for more information about simulating your design. See "Simulation Wizard Flow" on page 68 for information about creating a simulation project to run in Active-HDL.

For complete information about Active-HDL, see the *Active-HDL Online Documentation*, which is available from the Radiant Start Page or the Active-HDL Help menu.

Simulation Wizard

The Simulation Wizard enables you to create a simulation project for your design. To open Simulation Wizard, choose **Tools > Simulation Wizard** or click on the icon in the Radiant software toolbar. The wizard leads you through a series of steps that include selecting a simulation project name, location, specifying the simulator type, selecting the process stage to use, and selecting the source files. To learn more about the flow, view "Simulation Wizard Flow" on page 68.

Figure 78: Simulation Wizard



Source Template

Source Template provides templates for creating VHDL, Verilog, and constraint files. Templates increase the speed and accuracy of design entry. You can drag and drop a template directly to the source file. You can also create your own templates. For more information, see “Source Template” on page 29.

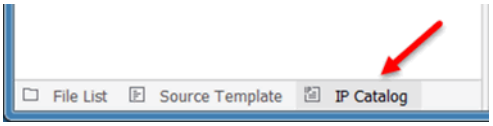
IP Catalog

IP Catalog is an easy way to use a collection of functional blocks from Lattice Semiconductor. There are two types of functional blocks available through IP Catalog: modules and IP. IP Catalog enables you to extensively customize these blocks. They can be created as part of a specific project or as a library for multiple projects.

- **Modules**: These basic, configurable blocks come with IP Catalog. They provide a variety of functions including I/O, arithmetic, memory, and more. Open IP Catalog to see the full list of what’s available.
- **IP**: Intellectual property (IP) are more complex, configurable blocks. They are accessible through IP Catalog, but they do not come with the tool. They must first be downloaded and installed in a separate step before they can be accessed from IP Catalog.

Overview of the IP Catalog Process Below are the basic steps of using IP Catalog modules and IP.

1. Open IP Catalog. IP Catalog is accessed via a tab at the lower left.of the Radiant software. Click the tab to view the list of available modules and IP.



2. Customize the module/IP. These modules and IP can be extensively customized for your design. The options may range from setting the width of a data bus to selecting features in a communications protocol. At a minimum you need to specify the design language to use for the output.
3. Generate the module/IP and bring its .ipx file into your project. Prior to generating the module/IP, select the option “Insert to project.” This will then automatically bring the .ipx file into your project after the generation step completes. If you do not select this option, add the .ipx file to your project after generation as you would with any other source file (such as a Verilog or VHDL file).
4. Instantiate the module/IP into the project’s design. An HDL instance template is created during generation to simplify this step.
5. IP Catalog modules and IP can be further modified or updated later. After the .ipx file has been added to the Radiant software project, it is visible in the project’s file list. Double-clicking the .ipx file brings up the module/IP’s configuration dialog box where changes can be made and the generation process repeated.

For more information on IP Catalog, see *Designing with Modules* in the Radiant online Help.

IP Packager

IP Packager allows external Intellectual Property (IP) developers, including third party IP providers and customers, to prepare and package IP in the Radiant IP format.

IP packages must contain certain files, including:

- Metadata file(s) (*.xml)
- RTL file(s) (*.v)
- Constraint template file (.ldc)
- Plugin file(s) (.py)
- Documentation files(s) (.htm, .html)
- License Agreement (*txt)

IP Packager is a standalone tool.

To run IP Packager:

- In Windows, go to the Windows Start menu and choose **Lattice Radiant Software > Accessories > IP Packager**.
- In Linux: from the ./<*Radiant install path*>/bin/lin64 directory, enter the following on a command line:

  ```
  ./ippackager
  ```

For more information on IP Packager, see R*unning Radiant IP Packager* in the Radiant online Help.

Common Tasks

The Radiant software gathers the many FPGA implementation tools into one central design environment. This gives you common controls for active tools, and it provides shared data between views.

Controlling Tool Views

Tool views are highly configurable in the Radiant software environment. You can detach a tool view to work with it as a separate window, or create tab groups to display two views side-by-side.

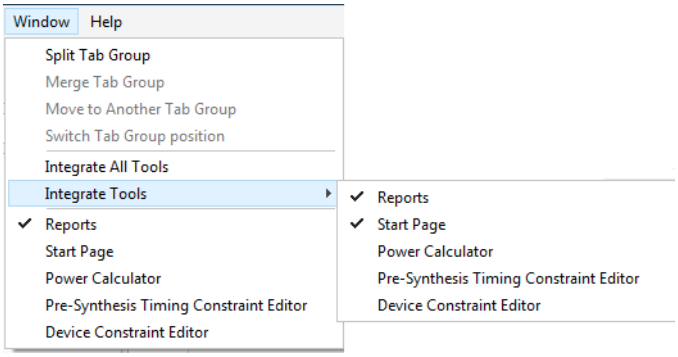
Detaching and Attaching a Tool View

Each integrated tool view contains a Detach button in the upper-right corner that allows you to work with the tool view as a separate window.

After a tool view is detached, the Detach button changes to an Attach button, which reintegrates the view into the Radiant main window.

You can detach as many tool views as desired. The Window menu keeps track of all open tool views and allows you to reintegrate one or all of them with the main window or detach any of them. Those that are already integrated are displayed with a check mark.

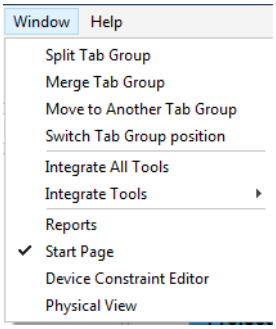
Figure 79: Window Integrate Tools Menu



Tab Grouping

The Radiant software allows you to split one or more active tools into a separate tab group. Use the Window menu or the toolbar buttons to create the tab group and control the display.

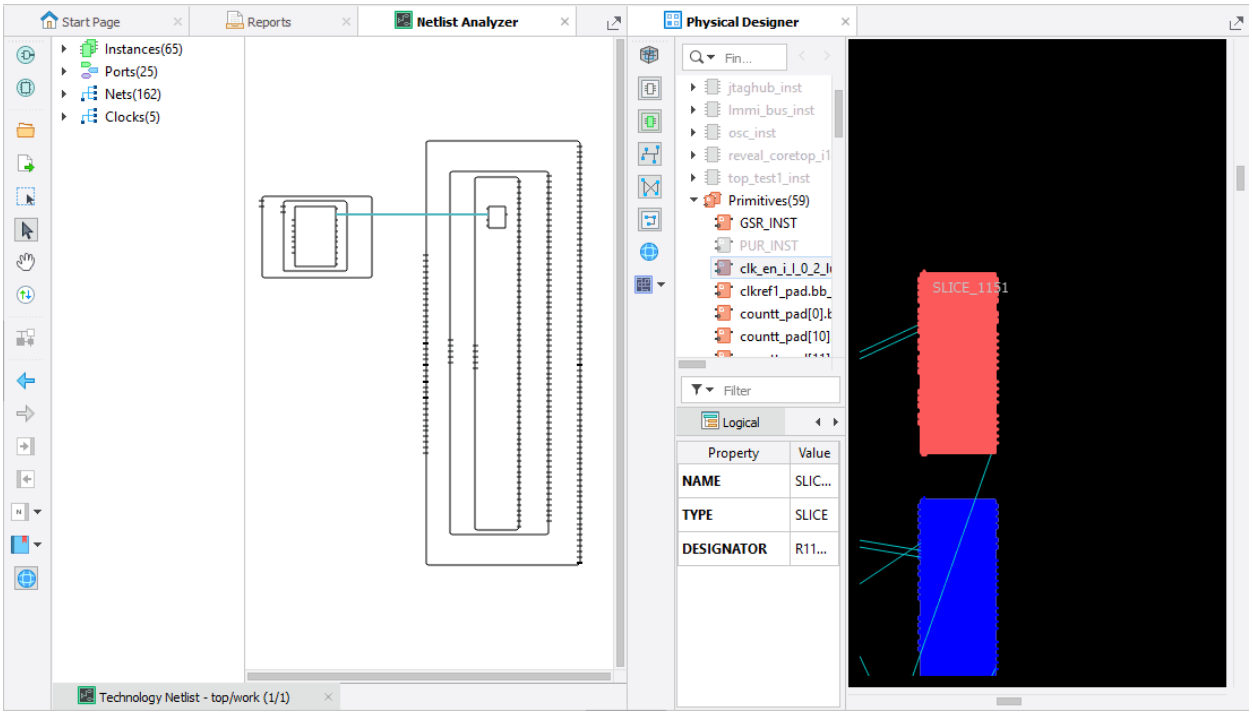
Figure 80: Tab Controls in View Menu



The Split Tab Group command separates the currently active tool into a separate tab group. Having two separate tab groups enables you to work with two tool views side-by-side. This is especially useful for dragging and dropping to make constraint assignments; for example, dragging a port from Netlist View to Package View in order to assign it to a pin.

Having two separate tab groups is also useful for examining the same data element in two different views, such as the Netlist Analyzer and Physical Designer layouts.

Figure 81: Split Tab Group with Side-by-Side Layout Views



Move an active tool view from one tab group to another by dragging and dropping it, or you can use the Move to Another Tab Group command.

To switch the positions of the two tab groups, click the Switch Tab Group Position command.

To merge the split tab group back into the main group, click the Merge Tab Group command.

Using Zoom Controls

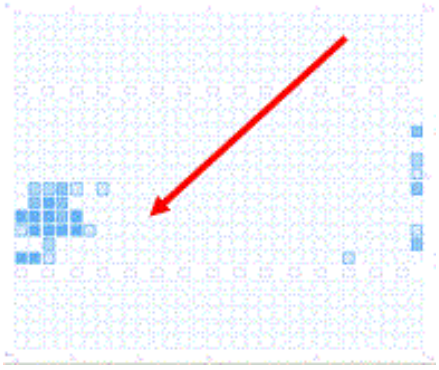
The Radiant software includes display zoom controls in the View toolbar. There are controls for increasing or reducing the scale of the view, fitting the display contents to the window view area, and fitting a selected area or object to the window view area.

Use the mouse to quickly zoom in, out or pan graphical views, such as Placement Mode and Routing Mode, by doing the following, as shown in Figure 82:

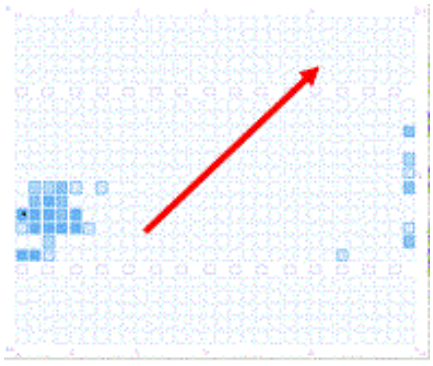
- Zoom In: press and hold the left mouse button while dragging the mouse down from upper right to left to zoom in.

- Zoom Out: press and hold the left mouse button while dragging the mouse up from lower left to right to zoom out.
- Zoom To: press and hold the left mouse button while dragging the mouse down from upper left to right to zoom into the box created.
- Zoom Fit: press and hold the left mouse button while dragging the mouse up from lower right to left to reset the diagram so it fits on screen.
- Pan: Click **Pan** ( ) and drag the mouse in any direction to move the diagram, or press and hold **Ctrl** and drag the mouse.

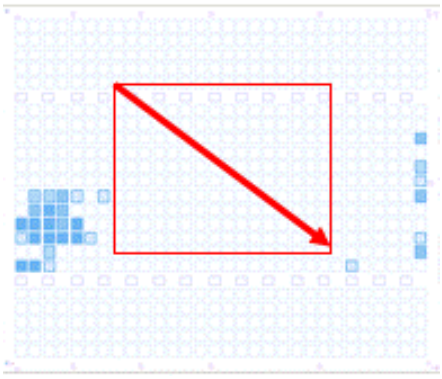
Figure 82: Zoom with Mouse



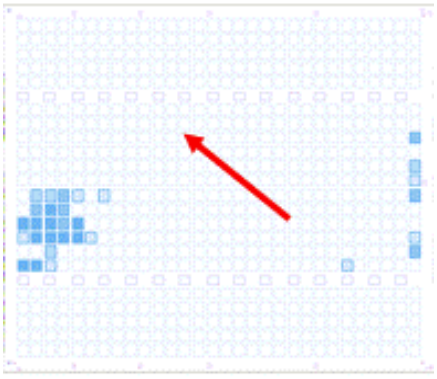
Zoom In



Zoom Out



Zoom Area



Zoom Fit

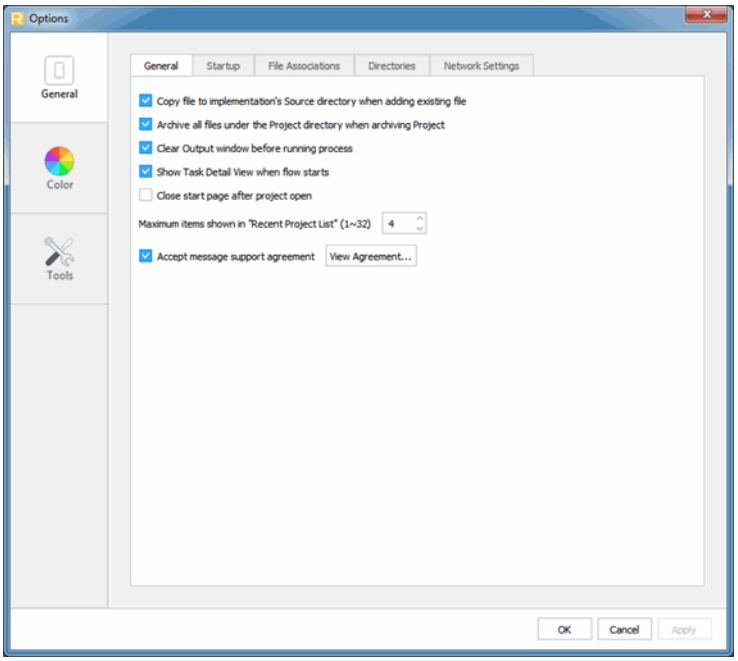
Displaying Tool Tips

When you place the cursor over a graphical element in a tool view, a tool tip appears with information on the element. The same information displayed in the tool tip will also be temporarily shown in the status bar on the lower left of the main window.

Setting Display Options

The Options dialog box, which is available from the Tools menu, enables you to specify general environment options as well as customize the display for the different tools. Tool options include selections for color, font, and other graphic elements.

Tool Options



For more information about Options, refer to “Environment and Tool Options” on page 184.

Command Line Reference Guide

This help guide contains information necessary for running the core design flow development from the command line. For tools that appear in the Radiant software graphical user interface, use Tcl commands to perform commands that are described in the “Tcl Command Reference Guide” on page 147.

Command Line Program Overview

Lattice FPGA command line programs can be referred to as the FPGA flow core tools. These are tools necessary to run a complete design flow and are used for tasks such as module generation, design implementation, design verification, and design configuration. This topic provides an overview of those tools, their functions, and provides links to detailed usage information on each tool.

Each command line program provides multiple options for specifying device information, applying special functions using switches, designating desired output file names, and using command files. The programs also have particular default behavior often precludes the need for some syntax, making commands less complex. See “Command Line General Guidelines” on page 98 and “Command Line Syntax Conventions” on page 99.

To learn more about the applications, usage, and syntax for each command line program, click on the hyperlink of the command line name in the section below.

Note

Many of the command line programs described in this topic are run in the background when using the tools you run in the Radiant software environment. Please also note that in some cases, command line tools described here are used for earlier FPGA architectures only, are not always recommended for command line use, or are only available from the command line.

Design Implementation Using Command Line Tools The table below shows all of the command line tools used for various design functions, their graphical user interface counterparts, and provides functional descriptions of each.

Table 1: The Radiant Software Core Design and Tool Chart

Design Function	Command Line Tool	Radiant Process	Description
Core Implementation and Auxiliary Tools			
Encryption	Encryption	Encrypt Verilog/VHDL files	Encrypts the input HDL source file with provided encryption key.
Synthesis	SYNTHESIS	Synthesis Design	Runs input source files through synthesis based on Lattice Synthesis Engine options set in Strategies.
Synthesis	Synpwrap	Synthesis Design	Used to manage Synopsys Synplify Pro synthesis programs.
Mapping	MAP	Map Design	Converts a design represented in logical terms into a network of physical components or configurable logic blocks.
Placement and Routing	PAR	Place & Route Design	Assigns physical locations to mapped components and creates physical connections to join components in an electrical network.
Static Timing Analysis	Timing	Post-Synthesis Timing Report, MAP Timing Report, Place & Route Timing Report	Generates reports that can be used for static timing analysis.
Back Annotation	Backanno	Tool does not exist in the Radiant software interface as process but employed in file export.	Distributes the physical design information back to the logical design to generate a timing simulation file.

Table 1: The Radiant Software Core Design and Tool Chart

Design Function	Command Line Tool	Radiant Process	Description
Bitstream Generation	BITGEN	Bitstream	Converts a fully routed physical design into configuration bitstream data.
Device Programming	PGRCMD	Device Programming	Downloads data files to an FPGA device.
IP Packager	IPPKG	IP Packager	IP author select files from disks and pack them into one IPK file.

See Also ▶"Command Line Data Flow" on page 97

- "Command Line General Guidelines" on page 98
- "Command Line Syntax Conventions" on page 99
- "Invoking Core Tool Command Line Programs" on page 101

Command Line Basics

This section provides basic instructions for running any of the core design flow development and tools from the command line.

Topics include:

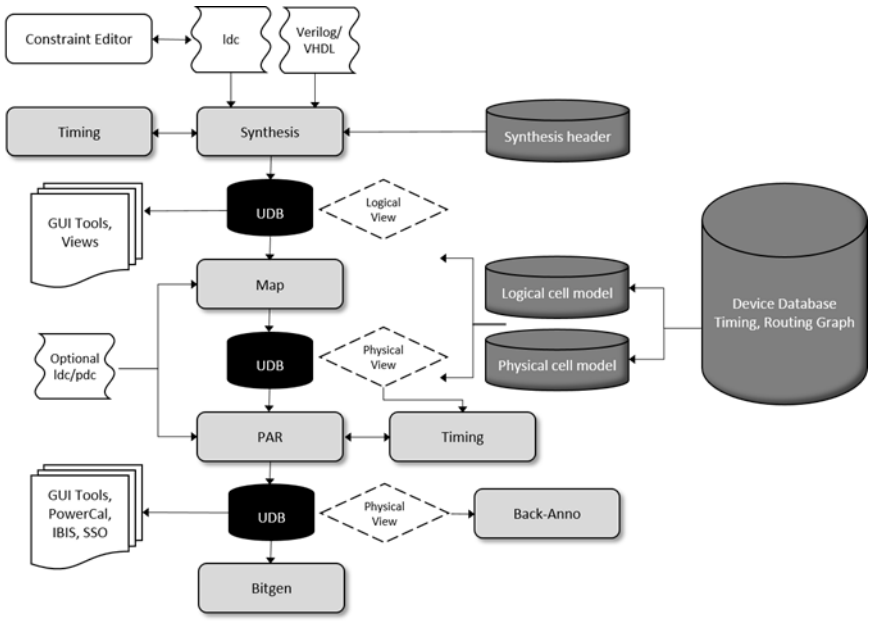
Command Line Data Flow

The following chart illustrates the FPGA command line tool data flow through a typical design cycle.

Command Line Tool Data Flow

See Also ▶"Command Line Reference Guide" on page 95

- "Command Line General Guidelines" on page 98



Command Line General Guidelines

You can run the FPGA family Radiant software design tools from the command line. The following are general guidelines that apply.

- Files are position-dependent. Generally, they follow the convention [options] <infile> <outfile> (although order of <outfile> and <infile> are sometimes reversed). Use the **-h** command line option to check the exact syntax; for example, **par -h**.
- For any Radiant software FPGA command line program, you can invoke help on available options with the **-h** or **-help** command. See "Invoking Core Tool Command Line Programs" on page 101 for more information
- Command line options are entered on the command line in any order, preceded by a hyphen (-), and separated by spaces.
- Most command line options are case-sensitive and must be entered in lowercase letters. When an option requires an additional parameter, the option and the parameter must be separated by spaces or tabs (i.e., **-l 5** is correct, **-l5** is not).
- Options can appear anywhere on the command line. Arguments that are bound to a particular option must appear after the option (i.e., **-f** <command_file> is legal; <command_file> **-f** is not).
- For options that may be specified multiple times, in most cases the option letter must precede each parameter. For example, **-b romeo juliet** is not acceptable, while **-b romeo -b juliet** is acceptable.
- If you enter the FPGA family Radiant software application name on the command line with no arguments and the application requires one or more arguments (**par,** for example), you get a brief usage message consisting of the command line format string.

- For any Radiant software FPGA command line program, you can store program commands in a command file. Execute an entire batch of arguments by entering the program name, followed by the **–f** option, and the command file name. This is useful if you frequently execute the same arguments each time you execute a program or to avoid typing lengthy command line arguments. See "Using Command Files" on page 142.

See Also "Command Line Reference Guide" on page 95

- "Invoking Core Tool Command Line Tool Help" on page 101
- "Command Line Syntax Conventions" on page 99
- "Using Command Files" on page 142
- "Command Line Data Flow" on page 97

Command Line Syntax Conventions

The following conventions are used when commands are described:

Table 2: Command Line Syntax Conventions

Convention	Meaning
( )	Encloses a logical grouping for a choice between sub-formats.
[ ]	Encloses items that are optional. (Do not type the brackets.) Note that <infile[**.udb**]> indicates that the .udb extension is optional but that the extension must be UDB.
{ }	Encloses items that may be repeated zero or more times.
|	Logical OR function. You must choose one or a number of options. For example, if the command syntax says **pan up | down | right | left** you enter **pan up** or **pan down** or **pan right** or **pan left**.
< >	Encloses a variable name or number for which you must substitute information.
, (comma)	Indicates a range for an integer variable.
- (dash)	Indicates the start of an option name.
:	The bind operator. Binds a variable name to a range.
bold text	Indicates text to be taken literally. You type this text exactly as shown (for example, "Type **autoroute -all -i 5** in the command area.") Bold text is also used to indicate the name of an EPIC command, a Linux command, or a DOS command (for example, "The **playback** command is used to execute the macro you created.").

Table 2: Command Line Syntax Conventions

Convention	Meaning
Italic text or text enclosed in angle brackets <>	Indicates text that is not to be taken literally. You must substitute information for the enclosed text. Italic text is also used to show a file directory path, for example, “the file is in the /cd/data/Radiant directory”).
Monospace	Indicates text that appears on the screen (for example, “File already exists.”) and text from Linux or DOS text files. Monospace text is also used for the names of documents, files, and file extensions (for example, “Edit the autoexec.bat file”

See Also ▶“Command Line Reference Guide” on page 95

- “Command Line General Guidelines” on page 98
- “Invoking Core Tool Command Line Tool Help” on page 101
- “Using Command Files” on page 142

Setting Up the Environment to Run Command Line

For Windows The environments for both the Radiant Tcl Console window or Radiant Standalone Tcl Console window (pnmainc.exe) are already set. You can start entering Tcl tool command or core tool commands in the console and the software will perform them.

When running the Radiant software from the Windows command line (via cmd.exe), you will need to add the following values to the following environment variables:

- PATH includes, for 64-bit

  `<Install_directory>\bin\nt64;<Install_directory>\ispfpga\bin\nt64`

  Example `<Install_directory>`:

  `c:\lscc\radiant\1.0\bin\nt64;c:\lscc\radiant\1.0\ispfpga\bin\nt64`

- FOUNDRY includes

  `set FOUNDRY= <Install_directory>\ispfpga`

For Linux On Linux, the Radiant software provides a similar standalone Tcl Console window (radiantc) that sets the environment. The user can enter Tcl commands and core tool commands in it.

If you do not use the Tcl Console window, you need to run "bash" to switch to BASH" first, then run the following command.

- For BASH (64-bit):

```
export bindir=<Install_directory>/bin/lin64
source $bindir/radiant_env
```

After setting up for either Windows or PC, you can run the Radiant software executable files directly. For example, you can invoke the Place and Route program by:

```
par test_map.udb test_par.udb
```

See Also ▶"Invoking Core Tool Command Line Programs" on page 101

- "Invoking Core Tool Command Line Tool Help" on page 101

Invoking Core Tool Command Line Programs

This topic provides general guidance for running the Radiant software FPGA flow core tools. Refer to "Command Line Program Overview" on page 95 to see what these tools include and for further information.

For any the Radiant software FPGA command line programs, you begin by entering the name of the command line program followed by valid options for the program separated by spaces. Options include switches (**-f**, **-p**, **-o**, etc.), values for those switches, and file names, which are either input or output files. You start command line programs by entering a command in the Linux™ or DOS™ command line. You can also run command line scripts or command files.

SeeTable 2 on page 99 for details and links to specific information on usage and syntax. You will find all of the usage information on the command line in the **Running FPGA Tools from the Command Line > Command Line Tool Usage** book topics.

See Also ▶"Command Line Reference Guide" on page 95

- "Command Line Syntax Conventions" on page 99
- "Invoking Core Tool Command Line Tool Help" on page 101
- "Setting Up the Environment to Run Command Line" on page 100
- "Using Command Files" on page 142

Invoking Core Tool Command Line Tool Help

To get a brief usage message plus a verbose message that explains each of the options and arguments, enter the FPGA family Radiant software application name on the command line followed by **-help** or **-h**. For example, enter **bitgen -h** for option descriptions for the **bitgen** program.

To redirect this message to a file (to read later or to print out), enter this command:

command_name -help | -h > filename

The usage message is redirected to the filename that you specify.

For those FPGA family Radiant software applications that have architecture-specific command lines (e.g., iCE UltraPlus), you must enter the application name, **-help** (or **-h**), and the architecture to get the verbose usage message specific to that architecture. If you fail to specify the architecture, you get a message similar to the following:

Use '<appname> -help <architecture>' to get detailed usage for a particular architecture.

See Also "Command Line Reference Guide" on page 95

- "Command Line Data Flow" on page 97
- "Command Line General Guidelines" on page 98
- "Command Line Syntax Conventions" on page 99
- "Setting Up the Environment to Run Command Line" on page 100
- "Using Command Files" on page 142

Command Line Tool Usage

This section contains usage information of all of the command line tools and valid syntax descriptions for each.

Topics include:

Running cmpl_libs.tcl from the Command Line

The cmpl_libs.tcl command allows you to perform simulation library compliation from the command line.

The following information is for running cmpl_libs.tcl from the command line using the tclsh application. The supported TCL version is 8.5 or higher.

If you don't have TCL installed, or you have an older version, perform the following:

- Add <*Radiant_install_path*>/tcltk/windows/BIN to the front of your PATH, and
- For Linux users only, add <*Radiant_install_path*>/tcltk/linux/bin to the front of your LD_LIBRARY_PATH

Note

The default version of TCL on Linux could be older and may cause the script to fail. Ensure that you have TCL version 8.5 or higher.

To check TCL version, type:

```
tclsh
% info tclversion
% exit
```

For script usage, type:

```
tclsh cmpl_libs.tcl [-h|-help|]
```

Notes

- If Modelsim/Questa is already in your PATH and preceding any Aldec tools, you can use:

  '-sim_path .' for simplification; '.' will be added to the front of your PATH.
- Ensure the FOUNDRY environment variable is set. If the FOUNDRY environment variable is missing, then you need to set it before running the script. For details, refer to “Setting Up the Environment to Run Command Line” on page 96.
- To execute this script error free, Questasim 10.4e or a later 10.4 version, or Questasim 10.5b or a later version should be used for compilation.

Check log files under <*target_path*> (defualt = .) for any errors, as follows:

- For Linux, type:

  ```
  grep -i error *.log
  ```

- For Windows, type:

  ```
  find /i "error" *.log
  ```

Subjects included in this topic:

- Running compl_lib.tcl
- Command Line Syntax
- cmpl_libs.tcl Options
- Examples

Running compl_lib.tcl cmpl_libs.tcl allows you to compile simulation libraries from the command line.

Command Line Syntax tclsh <*Radiant_install_path*>/cae_library/simulation/scripts/cmpl_libs.tcl -sim_path <sim_path> [-sim_vendor {mentor<default>}] [-device {ice40up|all<default>}] [-target_path <target_path>]

cmpl_libs.tcl Options The table below contains all valid options for cmpl_libs.tcl

Table 3: cmpl_libs.tcl Command Line Options

Option	Description
-sim_path <sim_path>	The -sim_path argument specifies the path to the simulation tool executable (binary) folder. This option is mandatory. Currently only Modelsim and Questa simulators are supported. NOTE: If <*sim_path*> has spaces, then it must be surrounded by " ". Do not use { }.
[-sim_vendor {mentor<default>}]	The -sim_vendor argument is optional, and intended for future use. It currently supports only Mentor Graphics simulators (Modelsim / Questa).
[-device {ice40up|all<default>}]	The -device argument specifies the Lattice FPGA device to compile simulation libraries for. This argument is optional, and the default is to compile libraries for all the Lattice FPGA devices.
[-target_path <target_path>]	The -target_path argument specifies the target path, where you want the compiled libraries and modelsim.ini file to be located. This argument is optional, and the default target path is the current folder. NOTES: (1) This argument is recommended if the current folder is tthe Radiant software's startup (binary) folder, or if the current folder is write-protected. (2) If <*target_path*> has spaces, then it must be surrounded by " ". Do not use { }.

Examples This section illustrates and describes a few examples of Simulation Libraries Compilation Tcl command.

Example 1 The following command will compile all the Lattice FPGA libraries for Verilog simulation, and place them under the folder specified by -target_path. The path to Modelsim is specified by -sim_path.

```
tclsh <c:/lscc/radiant/1.0/>/cae_library/simulation/script/
cmpl_libs.tcl -sim_path C:/modeltech64_10.0c/win64 -target_path
c:/mti_libs
```

See Also ▶"Command Line Program Overview" on page 95

Running HDL Encryption from the Command Line

Radiant software allows you to encrypt the individual HDL source files.

The tool supports encryption of Verilog HDL and VHDL files. Per command's execution, single source file is encrypted.

The HDL file can be partially or fully encrypted depending on pragmas' placements within the HDL file. To learn more about pragmas' placements, see "Defining Pragmas" on page 107.

Running HDL Encryption Before running the utility, you need to annotate the HDL file with the appropriate pragmas. Additionally, you may need to create a key file containing an encryption key. To view the key file's proper formating, see "Key File" on page 111.

Command Line Syntax

```
encrypt_hdl [-k <keyfile>] [-l language]
   [-o <output_file>] <input_HDL_file>
```

Encryption Option The table below contains descriptions of all valid options for HDL encryption.

Table 4: Encryption Command Line Options

Option	Description
-h(elp)	Print command help message.
-k <keyfile>	A key repository file. Depending on the location of the key, this option is required or optional. ▶ If the HDL source file contains no pragma, the key file is required. The tool encrypts the entire HDL file using all key sets declared in the key file. ▶ If the HDL source file contains only `begin` and `end` pragmas and no key pragmas, the key file is required. The tool encrypts the section between `begin` and `end` using all key sets declared in the key file. ▶ If the HDL source file contains the proper key pragma, key_keyowner, key_keyname, but the key file is missing the provided key_public_key, the tool fetches the first public key string matching the key_keyowner and key_keyname requirement in the key file If the HDL source file contains the proper definition of key, this option is not required. NOTE: If the same key name is defined in both, HDL source file and key.txt file, the key defined in HDL source file has a precedence.
-l <language>	Directive language, vhdl or verilog (default).
-o <output_file>	An encrypted HDL file. This is an optional field. If not defined during the encryption, the tool generates a new output file <input_file_name>_enc.v.

Examples This section illustrates and describes a few examples of HDL encryption using Tcl command.

Example1: This example shows a successful encryption of HDL file with default options. It is assumed that key is properly defined in HDL file. Since output file name was not specified, the tool generates an output file <file_name>_enc.v in the same directory as the location of the input file.

```
> encrypt_hdl -k source/impl_1/keys.txt -o top.v top.v
Options:
  Key repository file:     source/impl_1/keys.txt
  Directive language:     <not specified>, use verilog as default
  Output file:  top.v
Processed 2 envelopes.
```

Example2: This example shows a successful encryption of HDL file by generating a new output file.

```
> encrypt_hdl -k source/impl_1/keys.txt -o remote_files/top_v1_en.v remote_files/sources/top_v1_part.
Options:
  Key repository file:     source/impl_1/keys.txt
  Directive language:     <not specified>, use verilog as default
  Output file:  remote_files/top_v1_en.v
Processed 2 envelopes.
```

Example3: This example shows unsuccessful HDL encryption due to a missing key file. To correct this issue, the user must either define the appropriate key file key.txt or annotated the HDL file with appropriate pragmas. To correct the issue, define the key either in key.txt file or directly in HDL source file.

```
> encrypt_hdl -o remote_files/sources/top_v1_part_en.v remote_files/sources/top_v1_part.v
Options:
  Key repository file:     <not specified>
  Directive language:     <not specified>, use verilog as default
  Output file:  remote_files/sources/top_v1_part_en.v
ERROR - remote_files/sources/top_v1_part.v at line 88: missing key.
```

NOTE

A key is always required in the encryption tool while key file is optional. If the complete key: `key_keyowner`, `key_keyname`, `key_method`, and `key_public_key`, is defined within HDL source file, key file is not required.

For specific steps and information on how to encrypt HDL files in the Radiant software, refer to the following section in the Radiant software online help: **User Guides > Securing the Design**.

Defining Pragmas

Pragmas are used to specify the portion of the HDL source file that must be encrypted. Pragma' definition is compliant with IEEE 1735-2014 V1 standard.

Pragma syntax in Verilog HDL file:

```
`pragma protect <pragma's option>
```

Pragma synax in VHDL file:

```
`protect <pragma's option>
```

Table 5: List of available Pragma Options

Name	Available Values	Description
version	1 (default)	Specifies the current Radiant software encryption version.
author	string	Specifies the file creator.
author_info	string	Additional information you would like to include in file.
encoding	base64	The output format of processed data.
begin		The start point for data obfuscation.
end		The end point for data obfuscation.
key_keyowner	string	The key creator.
key_keyname	string	The RSA key name to specify the private key.
key_method	rsa	The cryptographic algorithm used for key obfuscation.

Table 5: List of available Pragma Options

Name	Available Values	Description
key_public_key		The RSA public key file name.
data_method	aes128-cbc aes256-cbc (default)	The AES encryption data method.

To encrypt HDL source file, encryption version, encoding type, and key specific pragmas must be defined in the HDL source file by HDL designer; only the content within the pragmas is encrypted.

NOTE

Multiple key sets can be declared in a single key file.

Example of Verilog source file marked with Pragmas:

```
// 3 bit counter with asynchronous reset
module count(c,clk,rst);

input clk,rst;
output [2:0]c;
reg [2:0]c;

`pragma protect version=1
`pragma protect author="<Your Name>"
`pragma protect author_info="<Your info>"
`pragma protect key_keyowner="Lattice Semiconductor"
`pragma protect key_keyname="LSCC_RADIANT_2"
`pragma protect key_method="rsa"
`pragma protect key_public_key
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA0EZKUUhbuB6vSsc7OhQJ
iNAWJR5unW/OWp/LFI71eAl3s9bOYE2OlKdxbai+ndIeo8xFt2btxetUzuR6Srvh
xR2Sj9BbW1QToo2u8JfzD3X7AmRvlwKRX8708DPo4LDHZMA3qh0kfDDWkp2Eausf
LzE2cVxgq7fy/bDhUeN8xKQCSKJ7aguG6kOI6ROoZz21ljzDLUQzhm2qYF8SpU1o
tD8/uw53wLfSuhR3MBOB++xcn2imvSLqdgHWuhX6CtZIx5CD4y8inCbcLy/0Qrf6
sdTN5SAg2OZhjeNdzmqSWqhL2JTDw+Ou2fWzhEd0i/HN0y4NMr6h9fNn8nqxRyE7
IwIDAQAB

//put a blank line above
`pragma protect data_method="aes256-cbc"

`pragma protect begin
always @(posedge clk or posedge rst)
     begin
        if (rst)
           c = 3'b000;
         else
           c = c + 1;
      end

`pragma protect end

endmodule
```

The encrypted file may contain multiple encrypted key sets.

Example of encrypted Verilog file:

```
// 3 bit counter with asynchronous reset
module count(c,clk,rst);

input clk,rst;
output [2:0]c;
reg [2:0]c;

//put a blank line above

`pragma protect begin_protected
`pragma protect version=1
`pragma protect author="<Your Name>"
`pragma protect author_info="<Your info>"
`pragma protect encrypt_agent="Radiant encrypt_hdl"
`pragma protect encrypt_agent_info="Radiant encrypt_hdl Version 1.0"

`pragma protect encoding=(enctype="base64", line_length=64, bytes=256)
`pragma protect key_keyowner="Lattice Semiconductor"
`pragma protect key_keyname="LSCC_RADIANT_2"
`pragma protect key_method="rsa"
`pragma protect key_block
gOatWm+1rVPboQIqaGf2gvNdUH/vTXzyRA4C+tdNxpqNWeeTXrTF+uIa21e9Io7S
K6ce3BVAXydtADq5Wy50EjcHzUi3Ym1eyEdfVn2pxCp+3csuiZSeNgbtYutonZjh
8ReQYzPqKt6fZvhZ1AqHiuuZhFUsZQFXqnT8IW4dQ7HWudREzx6jB7h+7vT+wJvH
N5kZMiHBFGRhiTePZ+yDOQwFVvwITezEoS099I8MoRGWl1U9kb4/Kenk96MIqE3W
lKAiQivIjXeWvLRqmOb0hNGRoOEYwjylY1pjq9GyelHoDC6czdyqOOWrunt//XNu
v/QtpeEe/co7o9arRtHbvw==

`pragma protect data_method="aes256-cbc"
`pragma protect encoding=(enctype="base64", line_length=64, bytes=176)
`pragma protect data_block
z+c6t504d6NkyrL5x6j6/raeaQmg0v9xmOQVaj3oq45SAsUIhGaihFVt+sS2kbNJ
/KBaqdciXAJHW8uRJtE/4nB+gZbVQsVnhRUlH/beksDnhdhWhNw/fcX6v6xptGwh
wQxsY+TjzT+pGC9r0EmAo2tndK63cWjHlg8hIZYnnw7yZAzv2OqylztSWFkdR9T4
mHclplFr96xb59oCRqbpQQgeESGgX9L1Yfd8j0ZXkSM=

`pragma protect end_protected

endmodule
```

Example of VHDL source file marked by Pragmas:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity top_test is

        port(
                cout : out std_logic_vector(7 downto 0);
                reset : in  std_logic;
                clk  : in  std_logic
            );
end top_test;

architecture top_test_arch of top_test is

signal count : std_logic_vector(7 downto 0);

begin

`protect version=1

`protect begin

  P100 : PROCESS(clk, reset)
  BEGIN
     IF (reset = '1') THEN
        count <= (others => '0');
     ELSE
        IF (clk'event and clk = '1') THEN
           count <= count + 1;
           cout <= count;
        END IF;
     END IF;
  END PROCESS;

`protect end


end top_test_arch;
```

Example of encrypted VHDL file:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity top_test is

        port(
                cout : out std_logic_vector(7 downto 0);
                reset : in  std_logic;
                clk  : in  std_logic
            );
end top_test;

architecture top_test_arch of top_test is

signal count : std_logic_vector(7 downto 0);

begin

`protect begin_protected
`protect version=1
`protect author="Lattice Semiconductor Corporation"
`protect author_info="Lattice Semiconductor Corporation"
`protect encrypt_agent="Radiant encrypt_hdl"
`protect encrypt_agent_info="Radiant encrypt_hdl Version 1.0"

`protect encoding=(enctype="base64", line_length=64, bytes=256)
`protect key_keyowner="Lattice Semiconductor"
`protect key_keyname="LSCC_RADIANT_2"
`protect key_method="rsa"
`protect key_block
Vi2YLO+x6rdARfQ9Dy5nkhsQDsmm1w6mdX+BGfDMCLHH9OVHbd1SmeHawCz0jXY8
ZfRK8VF/h1zOBmoosgY1pKd/Dpc5/4xkcEtnLzRbiXr1PhvP+tAFMMYrlFsqzJF/
0yoej8yT6mayYbd5mcEr5rzBmX29HuPBZbYr1ziac5IBpHZUa0mcwwhFnj5kz4OB
bVSqiay7v8ECP41vNzoRKrsTYKBOhiTa6UoG08ut2E4d8wJIXNBgZOuShYYzuOuv
1goVgwaRtFWrpTNXEmrZJPr/ikXRHTFzORpkDM7yNwGVTNJPMJ2aQde2wOi6EZWe
1NnRF4K2HK0Oz1NRbffIjQ==

`protect data_method="aes128-cbc"
`protect encoding=(enctype="base64", line_length=64, bytes=272)
`protect data_block
B7yNcwT9w4purXSxU1ln4f3rs1psxSPlV3pnWYipDj6rQSA7wniBxcC/laFebwKE
fvbKngTIn7N+W/1Den3kjpuznIvLy5cV/GTANFP0cWt9rnRrDCM5CYtNWqaMEZu7
o2QPLifpCvwEgygI0RO6NQS5frKo/jQLgOhf68+VpqFPozfrGAYI/YkEofh0foDH
9ScyO6grmJQCqtkqX2N3p6738N3iCFdKWJ6Udo5t++AT3YYeZZ7bprDN941k/BkU
YZij8fJx+qovJUWQmSUJYNnt2Vyoac4hsevXsFRxm439ssQtP4vHHEnraBSrHdVG
DJn0GqFID+tpC67tE61U2Y/yi18psFhZGs8jmuv9yGk=

`protect end_protected

end top_test_arch;
```

See Also ▶"Running HDL Encryption from the Command Line" on page 105

▶ "Key File" on page 111

Key File

The key repository file defines the cryptographic public key used for RSA encryption. In Radiant software, the key file contains Lattice public key. Additionally, it may contain some of the common EDA vendors public keys.

The Lattice public key file key.txt is located at <*Radiant_installed_directory*>/ispfpga/data/ folder. Aside of Lattice public key, the current version contains the public key for Synopsys, Aldec, and Cadence.

NOTE

If using Synplify Pro synthesis tool, both, the Lattice Public Key and the Synplify Pro Public Key must be defined in the key file. The Synplify Pro Public Key is used during the synthesis step to decrypt an encrypted design. The Lattice Public Key is used during the post-synthesis flow to decrypt an encrypted design.

A key file must contain properly declared pragmas such as `key_keyowner`, `key_keyname`, `key_method`, and `key_public_key` for each of the specified keys. The key value follows the `key_public_key` pragma.

The key file typically also contains the `data_method` pragma. It defines the algorithm used in data block encryption of HDL source file.

Example of a Key File:

```
// Use Verilog pragma syntax in this file
`pragma protect version=1
`pragma protect author="<Your Name>"
`pragma protect author_info="<Your info>"
`pragma protect key_keyowner="Lattice Semiconductor"
`pragma protect key_keyname="LSCC_RADIANT_2"
`pragma protect key_method="rsa"
`pragma protect key_public_key
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA0EZKUUhbuB6vSsc7OhQJ
iNAWJR5unW/OWp/LFI71eAl3s9bOYE2OlKdxbai+ndIeo8xFt2btxetUzuR6Srvh
xR2Sj9BbW1QToo2u8JfzD3X7AmRvlwKRX8708DPo4LDHZMA3qh0kfDDWkp2Eausf
LzE2cVxgq7fy/bDhUeN8xKQCSKJ7aguG6kOI6ROoZz21ljzDLUQzhm2qYF8SpU1o
tD8/uw53wLfSuhR3MBOB++xcn2imvSLqdgHWuhX6CtZIx5CD4y8inCbcLy/0Qrf6
sdTN5SAg2OZhjeNdzmqSWqhL2JTDw+Ou2fWzhEd0i/HN0y4NMr6h9fNn8nqxRyE7
IwIDAQAB

// Put a blank line above
// Add additonal public keys below this line

`pragma protect data_method="aes256-cbc"

// End of File
```

See Also ▶"Running HDL Encryption from the Command Line" on page 105

▶ "Defining Pragmas" on page 107

Running SYNTHESIS from the Command Line

The Lattice synthesis tool SYNTHESIS allows you to synthesize Verilog and VHDL HDL source files into netlists for design entry into the Radiant software environment. Based on your strategy settings you specify in the Radiant software, a synthesis project (.synproj) file is created and then used by SYNTHESIS using the -f option. The Radiant software translates strategy options into command line options described in this topic.

Verilog source files are passed to the program using the **-ver** option and VHDL source files are passed using the **-vhd** option. For mixed language

designs the language type is automatically determined by SYNTHESIS based on the top module of the design. For IP design, you must also specify IP location (**-ip_dir)**, IP core name (**-corename**), and encrypted RTL file name (**-ertl_file**).

Subjects included in this topic:

- Running SYNTHESIS
- Command Line Syntax
- SYNTHESIS Options
- Examples

Running SYNTHESIS SYNTHESIS will convert your input netlist (.v) file into a structural Verilog file that is used for the remaining mapping process.

- To run SYNTHESIS, type **synthesis** on the command line with valid options. A sample of a typical SYNTHESIS command would be as follows:

There are many command line options that give you control over the way SYNTHESIS processes the output file. Please refer to the rest of the subjects in this topic for more details. See examples.

Command Line Syntax **synthesis** [**-a** <arch>] [**-p** <device>] [**-sp** <performance_grade>] [**-t** <package_name>] [{**-path** <searchpath>}] [**-top** <top_module_name>] [**-ver** {<verilog_file.v>}] [**-lib** <libname>] [**-vhd** {<vhdl_file.vhd/vhdl>}] [**-udb** <udb_file.udb>] [**-hdl_param** < param_name param_value >] [**-vh2008**] [**-optimization_goal** <area | timing | balanced (default)>] [**-force_gsr** <auto(default) | yes | no>] [-**ramstyle** <auto(default) | distributed | block_ram(EBR) | registers>] [**-romstyle** <auto(default) | logic | EBR>] [**-output_edif** <filename.edf>] [**-output_hdl** <filename.v>] [**-sdc** <sdc_file.ldc>] [**-logfile** <synthesis_logfile>] [**-frequency** <target_frequency (default 200.0MHz (ICE40))>] [**-max_fanout** <max_fanout (default 1000)>] [**-bram_utilization** <bram_utilization (default 100%)>] [**-use_dsp** <0|1(default)>] [**-dsp_utilization** <dsp_utilization (default 100%)>][**-fsm_encoding_style** <auto(default) | one-hot | gray | binary>] [**-resolve_mixed_drivers** <0(default)|1>] [**-fix_gated_clocks** <0|1(default)] [**-use_carry_chain** <0|1(default)>] [**-carry_chain_length** <chain_length>] [**-use_io_insertion** <0|1(default)>] [**-use_io_reg** <0|1|auto(default)>] [**-resource_sharing** <0|1(default)>] [**-propagate_constants** <0|1(default)>] [**-remove_duplicate_regs** <0|1(default)>] [**-loop_limit** <max_loop_iter_cnt (default 1950)>] [**-twr_paths** <timing_path_cnt>] [**-dt**] [**-comp**] [**-syn**] [**-ifd**] [**-f** <project_file_name>]

SYNTHESIS Options The table below contains descriptions of all valid options for SYNTHESIS.

Table 6: SYNTHESIS Command Line Options

Option	Description
-a <arch>	Sets the FPGA architecture. This synthesis option must be specified and if the value is set to any unsupported FPGA device architecture the command will fail.
-p <device>	Specifies the device type for the architecture (optional).
-f <proj_file_name>	Specifies the synthesis project file name (.synproj). The project file can be edited by the user to contain all desired command line options.
-t <package_name>	Specifies the package type of the device.
-path <searchpath>	Add searchpath for Verilog “include” files (optional).
-top <top_module_name>	Name of top module (optional, but better to have to avoid ambiguity).
-lib <lib_name>	Name of VHDL library (optional).
-vhd <vhdl_file.vhd/vhdl>	Names of VHDL design files (must have, if language is VHDL or mixed language).
-ver <verilog_file.v>	Names of Verilog design files (must have, if language is Verilor, or mixed language).
-hdl_param <name, value>	Allows you to override HDL parameter pairs in the design file.
-optimization_goal <balanced (default)| area | timing>	The synthesis tool allows you to choose among the following optimization options: ▶ **balanced** balances the levels of logic. ▶ **area** optimizes the design for area by reducing the total amount of logic used for design implementation. ▶ **timing** optimizes the design for timing. The default setting depends on the device type. Smaller devices, such as ice40tp default to balanced.
-force_gsr <auto | yes | no>	Enables (**yes**) or disables (**no**) forced use of the global set/reset routing resources. When the value is auto, the synthesis tool decides whether to use the global set/reset resources.

Table 6: SYNTHESIS Command Line Options

Option	Description
-ramstyle <auto (default) | distributed | block_ram(EBR) | registers>	Sets the type of random access memory globally to *distributed*, *embedded block RAM*, or *registers*.The default is auto which attempts to determine the best implementation, that is, synthesis tool will map to technology RAM resources (EBR/Distributed) based on the resource availability. This option will apply a syn_ramstyle attribute globally in the source to a module or to a RAM instance. To turn off RAM inference, set its value to registers. ▶ **registers** causes an inferred RAM to be mapped to registers (flip-flops and logic) rather than the technology-specific RAM resources. ▶ **distributed** causes the RAM to be implemented using the distributed RAM or PFU resources. ▶ **block_ram (EBR)** causes the RAM to be implemented using the dedicated RAM resources. If your RAM resources are limited, for whatever reason, you can map additional RAMs to registers instead of the dedicated or distributed RAM resources using this attribute. ▶ **no_rw_check** (Certain technologies only). You cannot specify this value alone. Without no_rw_check, the synthesis tool inserts bypass logic around the RAM to prevent the mismatch. If you know your design does not read and write to the same address simultaneously, use no_rw_check to eliminate bypass logic. Use this value only when you cannot simultaneously read and write to the same RAM location and you want to minimize overhead logic.

Table 6: SYNTHESIS Command Line Options

Option	Description
-romstyle <auto (default) | logic | EBR>	Allows you to globally implement ROM architectures using *dedicated*, *distributed ROM*, or a *combination of the two* (auto). This applies the syn_romstyle attribute globally to the design by adding the attribute to the module or entity. You can also specify this attribute on a single module or ROM instance. Specifying a syn_romstyle attribute globally or on a module or ROM instance with a value of: ▶ **auto** allows the synthesis tool to choose the best implementation to meet the design requirements for performance, size, etc. ▶ **EBR** causes the ROM to be mapped to dedicated EBR block resources. ROM address or data should be registered to map it to an EBR block. If your ROM resources are limited, for whatever reason, you can map additional ROM to registers instead of the dedicated or distributed RAM resources using this attribute. Infer ROM architectures using a CASE statement in your code. For the synthesis tool to implement a ROM, at least half of the available addresses in the CASE statement must be assigned a value. For example, consider a ROM with six address bits (64 unique addresses). The case statement for this ROM must specify values for at least 32 of the available addresses.
-output_hdl <filename.v>	Specifies the name of the output Verilog netlist file.
-sdc <sdc_file.ldc>	Specifies a Lattice design constraint (.ldc) file input.
-loop_limit <max_loop_iter_cnt (default 1950)>	Specifies the iteration limits for “for” and “while” loops in the user RTL for loops that have the loop index as a variable and not a constant. The higher the loop_limit, the longer the run time. Also, for some designs, a higher loop limit may cause stack overflow during some of the optimizations during compile/synthesis. The default value is 1950. Setting a higher value may cause stack overflow during some of the optimizations during synthesis.
-logfile <synthesis_logfile>	Specifies the name of the synthesis log file in ASCII format. If you do not specify a name, SYNTHESIS will output a file named synthesis.log by default.
-frequency <target_frequency (default 200.0MHz (ICE40))>	Specifies the target frequency setting. Default frequency value is 200.0 MHz.
-max_fanout <value>	Specifies maximum global fanout limit to the entire design at the top level. Default value is 1000 fanouts.
-bram_utilization <value>	Specifies block RAM utilization target setting in percent of total vacant sites. Default is 100 percent.

Table 6: SYNTHESIS Command Line Options

Option	Description
-fsm_encoding_style <auto | one-hot | gray | binary>	Specifies One-Hot, Gray, or Binary style. The -fsm_encoding_style. Allows the user to determine which style is faster based on specific design implementation. Valid options are *auto*, *one-hot*, *gray*, and *binary*. The default value is auto, meaning that the tool looks for the best implementation.
-use_carry_chain <0|1>	Turns on (1) or off (0) carry chain implementation for adders. The 1 or true setting is the default.
-carry_chain_length <chain_length>	Specifies the maximum length of the carry chain.
-use_io_insertion <0|1>	Specifies the use of I/O insertion. The 1 or true setting is the default.
-use_io_reg <0|1|auto(default)>	Packs registers into I/O pad cells based on timing requirements for the target Lattice families. The value 1 enables and 0 disables (default) register packing. This applies it globally forcing the synthesis tool to pack all input, output, and I/O registers into I/O pad cells. NOTE: You can place the syn_useioff attribute on an individual register or port. When applied to a register, the synthesis tool packs the register into the pad cell, and when applied to a port, packs all registers attached to the port into the pad cell. The syn_useioff attribute can be set on a: ▶ top-level port ▶ register driving the top-level port ▶ lower-level port, only if the register is specified as part of the port declaration
-resource_sharing <0|1>	Specifies the resource sharing option. The 1 or true setting is the default.
-propagate_constants <0|1>	Prevents sequential optimization such as constant propagation, inverter push-through, and FSM extraction. The 1 or true setting is the default.
-remove_duplicate_regs <0|1>	Specifies the removal of duplicate registers. The 1 or true setting is the default.
-twr_paths <timing_path_cnt>	Specifies the number of critical paths.
-dt	Disables the hardware evaluation capability.
-udb <udb_file.udb>	
-ifd	Sets option to dump intermediate files. If you run the tool with this option, it will dump about 20 intermediate encrypted Verilog files. If you supply Lattice with these files, they can be decrypted and analyzed for problems. This option is good to for analyzing simulation issues.

Table 6: SYNTHESIS Command Line Options

Option	Description
-fix_gated_clocks <0|1(default)>	Allows you to enable/disable gated clock optimization. By default, the option is enabled.
-vh2008	Enables VHDL 2008 support.

Examples Following are a few examples of SYNTHESIS command lines and a description of what each does.

synthesis -a "ice40tp" -p itpa08 -t SG48 -sp "6" -mux_style Auto

-use_io_insertion 1

-sdc "C:/my_radiant_tutorial/impl1/impl1.ldc"

-path "C:/lscc/radiant/1.0/ispfpga/ice40tp/data" "C:/my_radiant_tutorial/impl1" "C:/my_radiant_tutorial"

-ver "C:/my_radiant_tutorial/impl1/source/LED_control.v"

"C:/my_radiant_tutorial/impl1/source/spi_gpio.v"

"C:/my_radiant_tutorial/impl1/source/spi_gui_led_top.v"

-path "C:/my_radiant_tutorial"

-top spi_gui_led_top

-output_hdl "LEDtest_impl1.vm"

See Also ▶"Command Line Program Overview" on page 95

Running Postsyn from the Command Line

The Postsyn process converts synthesized VM and integrates IPs into a completed design in UDB format for the remaining mapping process.

Command Line Syntax `postsyn [-w] [-a <architecture>] [-p <device>] [-t <package>] [-sp <performance>] [-ldc <ldc_file>] [-iplist <iplist_file>] [-o <output.udb>] [-keeprtl] [-top ] <input.vm>`

Table 7:

Option	Description
-h(elp)	Print command help message.
-w	Overwrite ouptut file.
-a	Target architecture name.
-p	Target device name.
-t	Target package name.
-sp	Target performance grade.
-oc	Target operating condition: commercial | industrial | automotive.
-ldc	Load LDC file.
-iplist	Load IP list file.
-o	Output UDB file.
-keeprtl	Keep RTL view if it exists in UDB file.
-top	Indicate that the input is for the top design.
<input.vm>	Input structural Verilog file.

See Also ▶"Command Line Program Overview" on page 95

Running MAP from the Command Line

The **Map Design** process in the Radiant software environment can also be run through the command line using the **map** program. The **map** program takes an input database (.udb) file and converts this design represented as a network of device-independent components (e.g., gates and flip-flops) into a network of device-specific components (e.g., PFUs, PFFs, and EBRs) or configurable logic blocks in the form of a Unified Database (.udb) file.

Subjects included in this topic:

- Running MAP
- Command Line Syntax
- MAP Options
- Examples

Running MAP MAP uses the database (.udb) file that was the output of the **Synthesis** process and outputs a mapped Unified Database (.udb) file with constraints embedded.

- To run MAP, type **map** on the command line with, at minimum, the required options to describe your target technology (i.e., architecture, device, package, and peformance grade), the input .udb along with the input .ldc file. The output .udb file specified by the **-o** option. That additional physical constraint file (*.pdc) can be applied optionally. A sample of a typical MAP command would be as follows:

  map counter_impl1_syn.udb impl1.pdc -o counter_impl1.udb

Note

The **-a** (architecture) option is not necessary when you supply the part number with the **-p** option. There is also no need to specify the constraint file here, but if you do, it must be specified after the input .udb file name. The constraint file automatically takes the name "**output**" in this case, which is the name given to the output .udb file. If the output file was not specified with the **-o** option as shown in the above case, **map** would place a file named input.udb into the current working directory, taking the name of the input file. If you specify the input.ldc file and it is not there, map will error out.

There are many command line options that give you control over the way MAP processes the output file. Please refer to the rest of the subjects in this topic for more details.

Command Line Syntax **map** [-h <arch>] <infile[.udb]> [<options>]

MAP Options The table below contains descriptions of all valid options for MAP.

Table 8: MAP Command Line Options

Option	Description
-h <arch>	Displays all of the available MAP command options for mapping to the specified architecture.
<infile[.udb]>	Specifies the output design file name in .udb format. The .udb extension is optional.
-inferGSR	GSR inferencing if applicable.
-o <name[.udb]>	Optional output design file .udb.
-mp <name[.mrp]>	Optional report file (.mrp).
-xref_sig	Report signal cross reference for renamed signals.
-xref_sym	Report symbol cross reference for renamed symbols.
-u	Unclip unused instances.

Examples Following are some examples of MAP command lines and a description of what each does.

Example 1 The following command maps an input database file named mapped.udb and outputs a mapped Unified Database file named mapped.udb.

map counter_impl1_syn.udb impl1.pdc -o counter_impl1.udb

See Also "Command Line Data Flow" on page 97

"Command Line Program Overview" on page 95

Running PAR from the Command Line

The **Place & Route Design** process in the Radiant software environment can also be run through the command line using the **par** program. The **par** program takes an input mapped Unified Database (.udb) file and further places and routes the design, assigning locations of physical components on the device and adding the inter-connectivity, outputting a placed and routed .udb file.

The Implementation Engine multi-tasking option available in Linux is explained in detail here because the option is not available for PCs.

Subjects included in this topic:

- Running PAR
- Command Line Syntax
- General Options
- Placement Options
- Routing Options
- PAR Explorer (-exp) Options
- Examples
- PAR Multi-Tasking Options

Running PAR PAR uses your mapped Unified Database (.udb) file that were the outputs of the **Map Design** process or the **map** program. With these inputs, **par** outputs a new placed-and-routed .udb file, a PAR report (.par) file, and a PAD (specification (.pad) file that contains I/O placement information.

- To run PAR, type **par** on the command line with at minimum, the name of the input .udb file and the desired name of the output .udb file. Design constraints from previous stages are automatically embedded in the input .udb file, however the par program can accept additional constraints with either a .pdc or .sdc file. A sample of a basic PAR command would be as follows:

par input.udb output.udb

There are many command line options that give you control over PAR. Please refer to the rest of the subjects in this topic for more details.

Command Line Syntax **par** [-w] [-n <iterations:0,100>] [-t <iteration:0,100>] [-stopzero] [-s <savecount:0,100>] [-m <nodelistfile>] [-cores <number of cores>] [-r] [-k] [-p] [-x] [-pack <density:0,100>] [-sp

<setupspeedgrade>] [-hsp <holdspeedgrade>] [-dh] [-hos] [-sort <method>] <infile> <outfile> [<pdcfile>]

Note

All filenames without special switches must be in the order <infile> <outfile> <pdcfile>. Options may exist in any order.

General Options

Table 9: General PAR Command Line Options

Option	Description
-f	Read par command line arguments and switches from file.
-w	Overwrite. Allows overwrite of an existing file (including input file).
-n	Number of iterations (seeds). Use "-n 0" to run until fully routed and a timing score of zero is achieved. Default: 1.
-t	Start at this placer cost table entry. Default is 1.
-stopzero	Stop running iterations once a timing score of zero is achieved.
-s	Save "n" best results for this run. Default: Save All.
-m	Multi task par run. File "<node list file>", contains a list of node names to run the jobs on.
-cores	Run multiple threads on the local machine. You can specify "<number of cores>" to run the jobs. For cases when the user specifies both -cores and -m with a valid node list file, PAR should apply both settings (merge). If the user repeats the host machine in the node list file, the settings in the node list file take precedence over the setting in -cores (for backwards compatibility).
-p	Don't run placement.
-r	Don't run router.
-k	Keep existing routing in input UDB file. Note: only meaningful when used with -p.
-x	Ignore timing constraints.
-pack	Set the packing density parameter. Default: auto.

Table 9: General PAR Command Line Options

Option	Description
-sp	Change performance grade for setup optimization. Default: Keep current performance grade.
-hsp	Change performance grade for hold optimization. Default: M.
-dh	Disable hold timing correction.
-hos	Prioritize hold timing correction over setup performance.
-sort	Set the sorting method for ranking multiple iterations. <method> "c" sorts by cumulative slack, "w" sorts by worst slack. Default: c.
<infile>	Name of input UDB file.
<outfile>	Name of output UDB file.

Table 10: PAR Placement Command Line Options

Option	Description
<pdcfile>	Name of optional constraint file. Note: the contents of <pdcfile> will overwrite all constraints saved in the input UDB file <infile>.

Examples Following are a few examples of PAR command lines and a description of what each does.

Example 1 The following command places and routes the design in the file input.udb and writes the placed and routed design to output.udb.

par input.udb output.udb

Example 2 The following command runs 20 place and route iterations. The iterations begin at cost table entry 5. Only the best 3 output design files are saved.

par -n 20 -t 5 -s 3 input.udb output.udb

Example 3 (Lattice FPGAs only) This is an example of **par** using the **-io** switch to generate .udb files that contain only I/O for viewing in the PAD Specification file for adjustment of ldc_set_location constraints for optimal I/O placement. You can display I/O placement assignments in the Radiant Spreadsheet View and choosing **View > Display IO Placement**.

par -io -w lev1bist.udb lev1bist_io.udb

Using the PAR Multi-Tasking (-m) Option This section provides information about environment setup, node list file creation, and step-by-step instructions for running the PAR Multi-tasking (**-m**) option from the command line. The PAR **-m** option allows you to use multiple machines (nodes) that are networked together for a multi-run PAR job, significantly reducing the total amount of time for completion. Before the multi-tasking option was developed, PAR could only run multiple jobs in a linear or serial fashion. The total time required to complete PAR was equal to the amount of time it took for each of the PAR jobs to run.

For example, the PAR command:

par -n 10 mydesign.udb output.udb

tells PAR to run 10 place and route passes (**-n 10**). It runs each of the 10 jobs consecutively, generating an output .udb file for each job, i.e., output_par.dir/5_1.udb, output_par.dir/5_2.udb, etc. If each job takes approximately one hour, then the run takes approximately 10 hours.

Suppose, however, that you have five nodes available. The PAR Multi-tasking option allows you to use all five nodes at the same time, dramatically reducing the time required for all ten jobs.

To run the PAR multi-tasking option from the command line:

1. First generate a file containing a list of the node names, one per line as in the following example:

```
# This file contains a profile node listing for a PAR multi
# tasking job.
[machine1]
SYSTEM = linux
CORENUM = 2
[machine2]
SYSTEM = linux
CORENUM = 2
Env = /home/user/setup_multipar.lin
Workdir = /home/user/myworkdir
```

   You must use the format above for the node list file and fill in all required parameters. Parameters are case insensitive.The node or machine names are given in square brackets on a single line.

   The **System** parameter can take linux or pc values depending upon your platform. However, the PC value cannot be used with Linux because it is not possible to create a multiple computer farm with PCs. **Corenum** refers to the number of CPU cores available. Setting it to zero will disable the node from being used. Setting it to a greater number than the actual number of CPUs will cause PAR to run jobs on the same CPU lengthening the runtime.

   The **Env** parameter refers to a remote environment setup file to be executed before PAR is started on the remote machine. This is optional. If the remote machine is already configured with the proper environment, this line can be omitted. To test to see if the remote environment is responsive to PAR commands, run the following:

```
ssh <remote_machine> par <par_option>
```

See the **System Requirements** section below for details on this parameter.

Workdir is the absolute path to the physical working directory location on the remote machine where PAR should be run. This item is also optional. If an account automatically changes to the proper directory after login, this line can be omitted. To test the remote directory, run the following,

```
ssh <remote_machine> ls <udb_file>
```

If the design can be found then the current directory is already available.

2. Now run the job from the command line as follows:

```
par -m nodefile_name -n 10 mydesign.udb output.udb
```

This runs the following jobs on the nodes specified.

```
Starting job 5_1 on node NODE1 at ...
Starting job 5_2 on node NODE2 at ...
Starting job 5_3 on node NODE3 at ...
Starting job 5_4 on node NODE4 at ...
Starting job 5_5 on node NODE5 at ...
```

As the jobs finish, the remaining jobs start on the nodes until all 10 jobs are complete. Since each job takes approximately one hour, all 10 jobs will complete in approximately two hours.

Note

If you attempt to use the multi-tasking option and you have specified only one placement iteration, PAR will disregard the **-m** option from the command and run the job in normal PAR mode. In this case you will see the following message:

WARNING - par: Multi task par not needed for this job. -m switch will be ignored.

System Requirements **ssh** must be located through the PATH variable. On Linux, the utility program's secure shell (**ssh**) and secure shell daemon (**sshd**) are used to spawn and listen for the job requests.

The executables required on the machines defined in the node list file are as follows:

- /bin/sh
- par (must be located through the PATH variable)

Required environment variable on local and remote machines are as follows:

- FOUNDRY (points at FOUNDRY directory structure must be a path accessible to both the machine from which the Implementation Engine is run and the node)
- LM_LICENSE_FILE (points to the security license server nodes)
- LD_LIBRARY_PATH (supports par path for shared libraries must be a path accessible to both the machine from which the Implementation Engine is run and the node)

To determine if everything is set up correctly, you can run the **ssh** command to the nodes to be used.

Type the following:

```
ssh <machine_name> /bin/sh -c par
```

If you get the usage message back on your screen, everything is set correctly. Note that depending upon your setup, this check my not work even though your status is fine.

If you have to set up your remote environment with the proper environment variables, you must create a remote shell environment setup file. An example of an ASCII file used to setup the remote shell environment would be as follows for ksh users:

```
export FOUNDRY=<install_directory>/ispfpga/bin/lin64
export PATH=$FOUNDRY/bin/lin64:$PATH
export LD_LIBRARY_PATH=$FOUNDRY/bin/lin:$LD_LIBRARY_PATH
64
```

For csh users, you would use the setenv command.

Screen Output When PAR is running multiple jobs and is not in multi-tasking mode, output from PAR is displayed on the screen as the jobs run. When PAR is running multiple jobs in multi-tasking mode, you only see information regarding the current status of the feature.

For example, when the job above is executed, the following screen output would be generated:

```
Starting job 5_1 on node NODE1
Starting job 5_2 on node NODE2
Starting job 5_3 on node NODE3
Starting job 5_4 on node NODE4
Starting job 5_5 on node NODE5
```

When one of the jobs finishes, this message will appear:

```
Finished job 5_3 on node NODE3
```

These messages continue until there are no jobs left to run.

See Also ▶"Implementing the Design" in the Radiant software online help

- "Command Line Data Flow" on page 97
- "Command Line Program Overview" on page 95

Running Timing from the Command Line

The **MAP Timing** and **Place & Route Timing** processes in the Radiant software environment can also be run through the command line using the **timing** program. Timing can be run on designs using the placed and routed Unified Design Database (.udb) and associated timing constraints specified in the design's (.ldc,.fdc, .sdc or .pdc) file or device constraints extracted from the design. Using these input files, **timing** provides static timing analysis and outputs a timing report file (.tw1/.twr).

Timing checks the delays in the Unified Design Database (.udb) file against your timing constraints. If delays are exceeded, Timing issues the appropriate timing error. See “Implementing the Design” in the Radiant software online help and associated topics for more information.

Subjects included in this topic:

- Running Timing
- Command Line Syntax
- Timing Options
- Examples

Running Timing Timing uses your input mapped or placed-and-routed Unified Design Database (.udb) file and associated constraint file to create a Timing Report.

- To run Timing, type **timing** on the command line with, at minimum, the names of your input .udb and sdc files to output a timing report (.twr) file. A sample of a typical Timing command would be as follows:

  timing design.udb (constraint is embedded in udb)

Note

The above command automatically generates the report file named design.twr which is based on the name of the .udb file.

There are several command line options that give you control over the way Timing generates timing reports for analysis. Please refer to the rest of the subjects in this topic for more details. See “Examples” on page 106.

Command Line Syntax timing <udb file name> [-sdc <sdc file name>] [-hld | -sethld] [-o <output file name>] [-v <integer>] [-endpoints <integer>] [-help]

Timing Options The following tables contain descriptions of all valid options for Timing.

Table 11: Compulsory Timing Command Line Options

Compulsory Option	Description
-db-file arg	design database file name.

Table 12: Optional Timing Command Line Options

Optional Option	Description
-endpoints arg (=10)	number of end points.
-u arg (=10)	number of unconstrained end points printed in the table.
-ports arg (=10)	number of top ports printed in the table.
-help	print the usage and exit.
-hld	hold report only.
-sp arg (=None)	Setup speed grade.
-hsp arg (=M)	Hold speed grade.
-rpt-file arg	timing report file name.
-o arg	timing report file name.
-alt_report	Diamond like report.
-report_sdc	Parsed file appears in report file.
-sdc-file arg	sdc file name.
-sethld	both setup and hold report.
-v arg (=10)	number of paths per constraint.
-time_through_async	Timer will time through async resets.
-iotime	compute the input setup/hold and clock to output delays of the FPGA.
-io_allspeed	Get worst IO results for all speed grades.
-pwrprd	Output clock information for PowerCalculator.
-nperend arg (=1)	Number of paths per end point.
-html	HTML format report.
-gui	Call from GUI.
-msg arg	Message log file.
-msgset arg	Message setting.

Examples Following are a few examples of Timing command lines and a description of what each does.

Example 1 The following command verifies the timing characteristics of the design named design1.udb, generating a summary timing report. Timing constraints contained in the file group1.prf are the timing constraints for the design. This generates the report file design1.twr.

timing design1.udb (constraint is embedded in udb)

Example 2 The following command produces a file listing all delay characteristics for the design named design1.udb. Timing constraints contained in the file group1.prf are the timing constraints for the design. The file output.twr is the name of the verbose report file.

timing -v design1.udb -o output.twr

Example 3 The following command analyzes the file design1.udb and reports on the three worst errors for each constraint in timing.prf. The report is called design1.twr.

timing -e 3 design1.udb

Example 4 The following command analyzes the file design1.udb and produces a verbose report to check on hold times on any FREQUENCY, CLOCK_TO_OUT, INPUT_SETUP and OFFSET constraints in the timing.prf file. With the output report file name unspecified here, a file using the root name of the .udb file (i.e., design1.twr) will be output by default.

timing -v -hld design1.udb

Example 5 The following command analyzes the file design1.udb and produces a summary timing report to check on both setup and hold times on any INPUT_SETUP and CLOCK_TO_OUT timing constraints in the timing.prf file. With the output report file name unspecified here, a file using the root name of the .udb file (i.e., design1.twr) will be output by default.

timing -sethld design1.udb

See Also ▶"Command Line Program Overview" on page 95

▶ "Command Line Data Flow" on page 97

Running Backannotation from the Command Line

The **Generate Timing Simulation Files** process in the Radiant software environment can also be run through the command line using the **backanno** program. The **backanno** program back-annotates physical information (e.g., net delays) to the logical design and then writes out the back-annotated design in the desired netlist format. Input to **backanno** is a Unified Database file (.udb) a mapped and partially or fully placed and/or routed design.

Subjects included in this topic:

- Running Backanno
- Command Line Syntax
- Backanno Options
- Examples

Running Backanno backanno uses your input mapped and at least partially placed-and-routed Unified Database (.udb) file to produce a back-annotated netlist (.v) and standard delay (.sdf) file. This tool supports all FPGA design architecture flows. Only Verilog netlist is generated.

- To run backanno, type **backanno** on the command line with, at minimum, the name of your input .udb file. A sample of a typical backanno command would be as follows:

  backanno backanno.udb

Note

The above command back annotates backanno.udb and generates a Verilog file backanno.v and an SDF file backanno.sdf. If the target files already exist, they will not be overwritten in this case. You would need to specify the **-w** option to overwrite them.

There are several command line options that give you control over the way backanno generates back-annotated netlists for simulation. Please refer to the rest of the subjects in this topic for more details.

Command Line Syntax (Verilog) **backanno** [**-w**] [**-pre** <prfx>] [**-sp** <grade>] [**-neg**] [**-pos**] [-sup] [**-min**] [**-x**] [**-fc**] [-slice] [-slice0] [-slice1] [-noslice] [**-t**] [**-dis** [<del>]] [**-m** [<limit>]] [**-u**] [**-i**] [**-nopur**] [**-l** <libtype>] [**-s** <separator>] [**-o** <verilog<<.v>>] [**-d** <delays[.sdf]>] [**-gui**] [**-msg** <msglogfile>] [**-msgset** <msgtypefile>] [<udbfile>]

Backanno Options The table below contains descriptions of all valid options for backanno.

Table 13: Backanno Options

Option	Description
-w	Overwrite the output files.
-sp <grade>	Override performance grade for backannotation.
-pre <prfx>	Prefix to add to module name to make them unique for multi-chip simulation.
-min	Override performance grade to minimum timing for hold check.
-dis <del>	Distribute routing delays by splitting the signal and inserting buffers. <del> is the maximum delay (in ps) between each buffer (1000ps by default).
-m <limit>	Shortens the block names to a given character limit in terms of some numerical integer value.

Table 13: Backanno Options

Option	Description
-u	Add pads for top-level dangling nets.
-neg	Negative setup/hold delay support. Without this option, all negative numbers are set to 0 in SDF.
-pos	Write out 0 for negative setup/hold time in SDF for SC.
-x	Generate x for setup/hold timing violation.
-i	Create a buffer for each block input that has interconnection delay.
-nopur	Do not write PUR instance in the backannotation netlist. Instead, user has to instantiate it in a test bench.
<type>	Netlist type to write out.
<libtype>	Library element type to use.
<netfile>	The name of the output netlist file. The extension on this file will change depending on which type of netlist is being written. Use -h <type>, where <type> is the output netlist type, for more specific information.
<udb file>	Input file '.udb '.

Examples Following are a few examples of backanno command lines and a description of what each does.

Example 1 The following command back annotates design.udb and generates a Verilog file design.vo and an SDF file design.sdf. If the target files exist, they will be overwritten.

```
backanno -w design.udb
```

Example 2 The following command back annotates design.udb and generates a Verilog file backanno.vo and an SDF file backanno.sdf. Any signal in the design that has an interconnection delay greater than 2000 ps (2 ns) will be split and a series of buffers will be inserted. The maximum interconnection delay between each buffer would be 2000 ps.

```
backanno -dis 2000 -o backanno design.udb
```

Example 3 The following command re-targets backannotation to peforcxmance grade -2, and puts a buffer at each block input to isolate the interconnection delay (ends at that input) and the pin to pin delay (starts from that input).

```
backanno -sp 2 -i design.udb
```

Example 4 The following command generates Verilog netlist and SDF files without setting the negative setup/hold delays to 0:

```
backanno -neg -n verilog design.udb
```

See Also "Command Line Program Overview" on page 95

"Command Line Data Flow" on page 97

Running Bit Generation from the Command Line

The **Bitstream** process in the Radiant software environment can also be run through the command line using the bit generation (**bitgen**) program. This topic provides syntax and option descriptions for usage of the **bitgen** program from the command line. The **bitgen** program takes a fully routed Unified Database (.udb) file as input and produces a configuration bitstream (bit images) needed for programming the target device.

Subjects included in this topic:

- Running BITGEN
- Command Line Syntax
- BITGEN Options
- Examples

Running BITGEN BITGEN uses your input, fully placed-and-routed Unified Database (.udb) file to produce bitstream (.bit, .msk, or .rbt) for device configuration.

- To run BITGEN, type **bitgen** on the command line with, at minimum, the **bitgen** command. There is no need to specify the input .udb file if you run **bitgen** from the directory where it resides and there is no other .udb present.

There are several command line options that give you control over the way BITGEN outputs bitstream for device configuration. Please refer to the rest of the subjects in this topic for more details.

iCE40UP Command Line Syntax **bitgen** [**-d**] [**-b**] [**-a**] [**-w**] [**-noebrinitq1**] [**-noebrinitq2**] [**-noebrinitq3**] [**-noheader**] [**-simbitmap**] [**-nvcm**] [**-freq** <frequency_bit_setting>] [**-spilowpower**] [**-warmboot**] [**-nvcmsecurity**] {**-g** <setting_value>} <infile> [<outfile>]

LIFCL Command Line Syntax **bitgen** [**-d**] [**-w**] [**-m** <format>] {**-site** <seirule>} {**-site** <seitype>} <infile> [<outfile>]

BITGEN Options The table below contains descriptions of all valid options for BITGEN.

Note

Many BITGEN options are only available for certain architectures. Please use the **bitgen** -h <*architecture*> help command to see a list of valid bitgen options for the particular device architecture you are targeting.

Table 14: iCE40UP BITGEN Command Line Options

Option	Description
-d	Disable DRC.
-**b**	Produce .rbt file (ASCII form of binary).
-**a**	Produce .hex file.
-**w**	Overwrite an existing output file.
-**freq** <frequency_bit_setting>	Can setup different frequency: 0 = slow, 1 = medium, 2 = fast. Depending on the speed of external PROM, this options adjusts the frequency of the internal oscillator used by the iCE40UP device during configuration. This is only applicable when the iCE40UP device is used in SPI Master Mode for configuration.
-**nvcm**	Produce NVCM file.
-**nvcmsecurity**	Set security. Ensures that the contents of the Non-Volatile Configuration Memory (NVCM) are secure and the configuration data cannot be read out of the device.
-**spilowpower**	SPI flash low power mode. Places the PROM in low-power mode after configuration. This option is applicable only when the iCE40UP device is used as SPI Master Mode for configuration.
-warmboot	Enable warm boot. Enables the Warm Boot functionality, provided the design contains an instance of the WARMBOOT primitive.
-noheader	Don’t include the bitstream header.
-noebrinitq0	Don’t include EBR initialization for quadrant 0.
-noebrinitq1	Don’t include EBR initialization for quadrant 1.
-noebrinitq2	Don’t include EBR initialization for quadrant 2.
-noebrinitq3	Don’t include EBR initialization for quadrant 3.
-g NOPULLUP:ENABLED	No IO pullup. Removes the pullup on the unused I/Os, except Bank 3 I/Os which do not have pullup.

Table 14: iCE40UP BITGEN Command Line Options

Option	Description
-h <architecture> or **-help** <architecture>	Display available BITGEN command options for the specified architecture. The **bitgen -h** command with no architecture specified will display a list of valid architectures.
<infile>	The input post-PAR design database file (.udb).
<outfile>	The output file. If you do not specify an output file, BITGEN creates one in the input file's directory If you specify -b, the extension is .rbt. If you specify –a, the extension is .hex. If you specify –nvcm, the extension is .nvcm. Otherwise the extension is .bin. A report (.bgn) file containing all of BITGEN's output is automatically created under the same directory as the output file.

Table 15: LIFCL BITGEN Command Line Options

Option	Description
-d	Disable DRC.
-**w**	Overwrite an existing output file.
-**m** <format>	Create "mask" and "readback" files. Valid formats are: 0: Output files in ASCII 1: Output files in binary.
-**g** <opt:val>	Set option to value, options are (First is default): CfgMode Disable, Flowthrough, Bypass RamCfg Reset, NoReset DONEPHASE T3, T2, T1, T0 GOEPHASE T1, T3, T2 GSRPHASE T2, T3, T1 GWEPHASE T2, T3, T1 ES Yes, No.
-h <architecture> or **-help** <architecture>	Display available BITGEN command options for the specified architecture. The **bitgen -h** command with no architecture specified will display a list of valid architectures.

Table 15: LIFCL BITGEN Command Line Options

Option	Description
<infile>	The input post-PAR design database file (.udb).
<outfile>	The output file. If you do not specify an output file, BITGEN creates one in the input file's directory If you specify -b, the extension is .rbt. If you specify –a, the extension is .hex. If you specify –nvcm, the extension is .nvcm. Otherwise the extension is .bin. A report (.bgn) file containing all of BITGEN's output is automatically created under the same directory as the output file.

Example The following command tells **bitgen** to overwrite any existing bitstream files with the **-w** option, prevents a physical design rule check (DRC) from running with **-d**, specifies a raw bits (.rbt) file output with **-b**. Notice how these three options can be combined with the **-wdb** syntax.

```
bitgen -wdb <design.udb>
```

See Also ▶"Command Line Program Overview" on page 95

- "Command Line Data Flow" on page 97

Running Programmer from the Command Line

You can run Programmer from the command line. The **PGRCMD** command uses a keyword preceded by a hyphen for each command line option.

Running PGRCMD PGRCMD allows you to download data files to an FPGA device.

- To run PGRCMD, type **pgrcmd** on the command line with, at minimum, the **pgrcmd** command.

There are several command line options that give you control over the way PGRCMD programs devices. Please refer to the rest of the subjects in this topic for more details.

Command Line Syntax The following describes the PGRCMD command line syntax:

pgrcmd [-**help**] [**-infile** <input_file_path>] [**-logfile** <log_file_path>] [**-cabletype** <cable>]

-cabletype

lattice [**-portaddress** < 0x0378 | 0x0278 | 0x03bc | 0x<*custom address*> >]

usb [**-portaddress** < EZUSB-0 | EZUSB-1 | ... | EZUSB-15 >]

usb2 [**-portaddress** < FTUSB-0 | FTUSB-1 | ... | FTUSB-15 >]

TCK [**0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10**]

PGRCMD Options The following are PGRCMD options.

Help (Optional)

Option	Description
-help or **-h**	Displays the Programmer command line options.

Input File (required)

Option	Description
-infile *filename*.xcf	Specifies the chain configuration file (.xcf). If the file path includes spaces, enclose the path in quotes.

Log File (optional)

Option	Description
-logfile *logfilename*.log	Specifies the location of the Programmer log file.

Cable Type (optional)

Option	Description
-cabletype lattice	Lattice HW-DLN-3C parallel port programming cable (default).
-cabletype usb	Lattice HW-USBN-2A USB port programming cable.
-cabletype usb2	Lattice FHW-USBN-2B (FTDI) USB programming cable and any FTDI based demo boards.

Parallel Port Address (optional)

Option	Description
-portaddress 0x0378	LPT1 parallel port (default)
-portaddress 0x0278	LPT2 parallel port
-portaddress 0x03BC	LPT3 parallel port
-portaddress 0x<*custom address*>	Custom parallel port address

This option is only valid with parallel port cables.

USB Port Address (optional)

Option	Description
-portaddress EZUSB-0 ... EZUSB-15	HW-USBN-2A USB cable number 0 through 15
-portaddress FTUSB-0 ... FTUSB-15	FTDI based demo board or FTDI USB2 cable number 0 through 15

Default is EZUSB-0 and FTUSB-0. Only valid with the USB port cables.

FTDI Based Demo Board or Cable Frequency Control (optional)

Option	Description
-TCK 0, 1, 2, 3, 4, ,5 ,6 7, 8, 9, 10	0 = 30 Mhz 1 = 15 Mhz (default) 2 = 10 Mhz 3 = 7.5 Mhz 4 = 6 Mhz 5 = 5 Mhz 6 = 4 Mhz 7 = 3 Mhz 8 = 2 Mhz 9 = 1 Mhz 10 = 900 Khz

Calculation formula for USB-2B (2232H FTDI USB host chip): Frequency = 60 MHz / (1 + ClockDivider) *2

Calculation formula for USB-2B (2232D FTDI USB host chip): Frequency = 12 MHz / (1 + ClockDivider) *2

Only applicable for FTDI based demo boards or programming cable.

Return Codes

Code	Definition
0	Success
-1	Log file error
-2	Check configuration setup error
-3	Out of memory error
-4	NT driver error

Code	Definition
-5	Cable not detected error
-6	Power detection error
-7	Device not valid error
-8	File not found error
-9	File not valid error
-10	Output file error
-11	Verification error
-12	Unsupported operation error
-13	File name error
-14	File read error
-17	Build SVF file error
-18	Build VME file error
-19	Command line syntax error

Examples The following is a PGRCMD example.

```
pgrcmd -infile c:\test.xcf
```

See Also ▶"Command Line Data Flow" on page 97

"Command Line Program Overview" on page 95

Running Various Utilities from the Command Line

The command line utilities described in this section are not commonly used by command line users, but you often see them in the auto-make log when you run design processes in the Radiant software environment. Click each link below for its function, syntax, and options.

Note

For information on commonly-used FPGA command line tools, see "Command Line Basics" on page 97.

Synpwrap

The **synpwrap** command line utility (wrapper) is used to manage Synplicity Synplify and Synplify Pro synthesis programs from the Radiant software environment processes: **Synplify Synthesize Verilog File** or **Synplify Synthesize VHDL File**.

The **synpwrap** utility can also be run from the command line to support a batch interface. For details on Synplify see the Radiant software online help. The **synpwrap** program drives **synplify_pro** programs with a Tcl script file containing the synthesis options and file list.

Note

This section supersedes the “Process Optimization and Automation” section of the *Synplicity Synplify and Synplify Pro for Lattice User Guide*.

This section illustrates the use of the **synpwrap** program to run Synplify Pro for Lattice synthesis scripts from the command line. For more information on synthesis automation of Synplify Pro, see the “User Batch Mode” section of the *Synplicity Synplify and Synplify Pro for Lattice User Guide*.

If you use Synplify Pro, the Lattice OEM license requires that the command line executables **synplify_pro** be run by the Lattice “wrapper” program, **synpwrap**.

Command Line Syntax **synpwrap** [**-log** <*log_file*>] [**-nolog**] [**-int** <*command_file*>] [**-gui**] [**-int** <*project_file*> | **-prj** <*project_file*>] [**-dyn**] [**-notoem**] [**-oem**] [**-notpro**] [**-pro**] [**-rem**] [**-scriptonly** <*script_file*>] **-e** <*command_file*> **-target** <*device_family*> **-part** <*device_name*> [**-options** <*arguments*>]

Table 16: SYNPWRAP Command Line Options

Option	Description
-log <*log_file*>	Specifies the log file name.
-nolog	Does not print out the log file after the process is finished.
-options <*arguments*>	Passes all arguments to Synplify/Pro. Ignores all other options except -notoem/-oem and -notpro/-pro. The -options switch must follow all other synpwrap options.
-prj <*project_file*>	Runs Synplify or Synplify Pro using an external prj Tcl file instead of the Radiant software command file.
-rem	Does not automatically include Lattice library files.
-e <*command_file*>	Runs the batch interface based on a Radiant software generated command file. The synpwrap utility reads <project>.cmd with its command line to obtain user options and creates a Tcl script file.
-gui	Invokes the Synplify or Synplify Pro graphic user interface.
-int <*command_file*>	Enables the interactive mode. Runs Synplify/Pro UI with project per command file.
-dyn	Brings the Synplify installation settings in the Radiant software environment.
-notoem	Does not use the Lattice OEM version of Synplify or Synplify Pro.

Table 16: SYNPWRAP Command Line Options

Option	Description
-oem	Uses the Lattice OEM version of Synplify or Synplify Pro.
-notpro	Does not use the Synplify Pro version.
-pro	Uses the Synplify Pro version.
-target <*device_family*>	Specifies the device family name.
-part <*device_name*>	Specifies the device. For details on legal <device_name> values.
-scriptonly <*script_file*>	Generates the Tcl file for Synplify or Synplify Pro. Does not run synthesis.

Example Below shows a synpwrap command line example.

```
synpwrap -rem -e prep1 -target iCE40UP
```

See Also ▶"Command Line Program Overview" on page 95

- "Command Line Data Flow" on page 97

IP Packager

The IP Packager (ippkg) tool can be run from the command line, allowing IP developers to select files from disks and pack them into one IPK file.

The process of IP packager is as following:

- IP author prepares metadata files, RTL files, HTML files, etc (all files of a Soft IP).
- IP Packager GUI provides UI for IP author to select files from the disk, and call IP Packaging engine to pack them into an IPK file.
- IP Packaging engine encrypts RTL files if IEEE P1735-2014 V1 pragmas are specified in RTL source

Command Line Syntax **ippkg** [**-h**] (**-metadata** METADATA_FILE | **-metadata_files** METADATA_LIST_NAME) (**-rtl** RTL_FILE | **-rtl**_files RTL_LIST_NAME) [**-plugin** PLUGIN_FILE] [**-ldc** LDC_FILE] [**-testbench** TESTBENCH_FILE | **-testbench_files** TESTBENCH_LIST_NAME] (**-help_file** HELP_FILE | **-help_files** HELP_LIST_NAME) [**-o** OUTPUT_ZIP_FILE] [**-key_file** KEY_FILE] [**--force-run**]

Table 17: IPPKG Command Line Options

Option	Description
-name	Specify the IP name.
-metadata	The file name will be fixed to 'metadata.xml.

Table 17: IPPKG Command Line Options

Option	Description
-metadata_files	Location of the file which stores the metadata files. One line is a file path in specified file. Must have a file named metadata.xml.
-rtl	Specify the IP RTL file.
-rtl_files	One line is a file path in specified file.
-plugin	The file name will be fixed to 'plugin.py'.
-ldc	Specify the LDC file.
-testbench	Specify the testbench file.
-testbench_files	One line is a file path in specified file.
-help_file	Specify the help file, must be <path>/introduction.html.
-help_files	One line is a file path in specified file.
-license_file	Specify the license file.
-o	Specify the output zip file.
-key_file	Specify the key file to encrypt the RTL files.
--force-run	Force program to run regardless of errors.

Example The following is an ippkg command line example:

```
ippkg –metadata c:/test/test.xml –rtl_files c:/test/rtl_list –
help_file c:/test/introduction.html
```

See Also ▶"Command Line Program Overview" on page 95

▶ "Command Line Data Flow" on page 97

ECO Editor

The ECO Editor tool can be run from the command line too.

ECO Editor is also able to dump the ECO TCL commands which user acted in GUI view without saving any UDB file.

In the meanwhile, we will have one non-GUI ECO engine tool, it accepts the dumped TCL script file with a UDB file and output a new UDB file.

User can set 'Place & Route design' milestone post-script by Tcl command prj_set_postscript par <eco.tcl>, then Radiant flow runs the ECO Tcl script automatically after running place & route.

Command Line Syntax **ecoc** [**-s** <*script_file*>] [**-o** <*output.udb*>] <in*put.udb*>**]**

Table 18: ECO Editor Command Line Options

Option	Description
-s	ECO Tcl script file.
-o	Output UDB file.
<*input.udb*>	Input UDB file.

Example The following is an ecoc command line example:

```
ecoc -s mem.tcl ebr_test_impl_1.udb
```

See Also ▶"Command Line Program Overview" on page 95

▶ "Command Line Data Flow" on page 97

Using Command Files

This section describes how to use command files.

Creating Command Files The command file is an ASCII file containing command arguments, comments, and input/output file names. You can use any text editing tool to create or edit a command file, for example, **vi**, **emacs**, **Notepad**, or **Wordpad**.

Here are some guidelines when you should observe when creating command files:

- Arguments (executables and options) are separated by space and can be spread across one or more lines within the file.
- Place new lines or tabs anywhere white space would otherwise be allowed on the Linux or DOS command line.
- Place all arguments on the same line, or one argument per line, or any combination of the two.
- There is no line length limitation within the file.
- All carriage returns and other non-printable characters are treated as space and ignored.
- Comments should be preceded with a # (pound sign) and go to the end of the line.

Command File Example This is an example of a command file:

```
#command line options for par for design mine.udb
-a -n 10
-w
-l 5
-s 2 #will save the two best results
/home/users/jimbob/designs/mine.udb
```

```
#output design name
/home/users/jimbob/designs/output.dir
#use timing constraint file
/home/users/jimbob/designs/mine.prf
```

Using the Command File The –f Option Use the **–f** option to execute a command file from any command line tool. The **–f** option allows you to specify the name of a command file that stores and then executes commonly used or extensive command arguments for a given FPGA command line executable tool. You can then execute these arguments at any time by entering the Linux or DOS command line followed by the name of the file containing the arguments. This can be useful if you frequently execute the same arguments each time you perform the command, or if the command line becomes too long. This is the recommended way to get around the DOS command line length limitation of 127 characters. (Equivalent to specifying a shell Options file.)

The **–f** indicates fast startup, which is performed by not reading or executing the commands in your .cshrc | .kshrc | .shrc (C-shell, Korn-shell, Bourne-shell) file. This file typically contains your path information, your environment variable settings, and your aliases. By default, the system executes the commands in this file every time you start a shell. The **–f** option overrides this process, discarding the 'set' variables and aliases you do not need, making the process much faster. In the event you do need a few of them, you can add them to the command file script itself.

Command File Usage Examples You can use the command file in two ways:

- To supply all of the command arguments as in this example:

par -f <command_file>

where:

<command_file> is the name of the file containing the command line arguments.

- To insert certain command line arguments within the command line as in the following example:

par -i 33 -f placeoptions -s 4 -f routeoptions design_i.udb design_o.udb

where:

placeoptions is the name of a file containing placement command arguments.

routeoptions is the name of a file containing routing command arguments.

Using Command Line Shell Scripts

This topic discusses the use of shell scripts to automate either parts of your design flow or entire design flows. It also provides some examples of what you can do with scripts. These scripts are Linux-based; however, it is also possible to create similar scripts called batch files for PC but syntax will vary in the DOS environment.

Creating Shell Scripts A Linux shell script is an ASCII file containing commands targeted to a particular shell that interprets and executes the commands in the file. For example, you could target Bourne Shell (**sh**), C-Shell (**csh**), or Korn Shell (**ksh**). These files also can contain comment lines that describe part of the script which then are ignored by the shell. You can use any text editing tool to create or edit a shell script, for example, **vi** or **emacs**.

Here are some guidelines when you should observe when creating shell scripts:

- It is recommended that all shell scripts with “#!” followed by the path and name of the target shell on the first line, for example, #!/bin/ksh. This indicates the shell to be used to interpret the script.
- It is recommended to specify a search path because oftentimes a script will fail to execute for users that have a different or incomplete search path. For example:

  PATH=/home/usr/lsmith:/usr/bin:/bin; export PATH
- Arguments (executables and options) are separated by space and can be spread across one or more lines within the file.
- Place new lines or tabs anywhere white space would otherwise be allowed on the Linux command line.
- Place all arguments on the same line, or one argument per line, or any combination of the two.
- There is no line length limitation within the file.
- All carriage returns and other non-printable characters are treated as space and ignored.
- Comments are preceded by a # (pound sign) and can start anywhere on a line and continue until the end of the line.
- It is recommended to add exit status to your script, but this is not required.

```
# Does global timing meet acceptable requirement range?
    if [ $timing -lt 5  -o  $timing -gt 10 ]; then
         echo 1>&2 Timing \"$timing\" out of range
         exit 127
    fi
    etc...
    # Completed, Exit OK
    exit 0
```

Advantages of Using Shell Scripts Using shell scripts can be advantageous in terms of saving time for tasks that are often used, in

reducing memory usage, giving you more control over how the FPGA design flow is run, and in some cases, improving performance.

Scripting with DOS Scripts for the PC are referred to as batch files in the DOS environment and the common practice is to ascribe a .bat file extension to these files. Just like Linux shell scripts, batch files are interpreted as a sequence of commands and executed. The COMMAND.COM or CMD.EXE (depending on OS) program executes these commands on a PC. Batch file commands and operators vary from their Linux counterparts. So, if you wish to convert a shell script to a DOS batch file or vice-versa, we suggest you find a good general reference that shows command syntax equivalents of both operating systems.

Examples The following example shows running design “counter” on below device package

Architecture: iCE40UP

Device: iCE40UP3K

Package: UWG30

Performance: Worst Case

```
Command 1: logic synthesis
synthesis -f counter_impl1_lattice.synproj
     which the *.synproj contains
-a "iCE40UP"
-p iCE40UP3K
-t UWG30
-sp "Worst Case"
-optimization_goal Area
-bram_utilization 100
-ramstyle Auto
-romstyle auto
-dsp_utilization 100
-use_dsp 1
-use_carry_chain 1
-carry_chain_length 0
-force_gsr Auto
-resource_sharing 1
-propagate_constants 1
-remove_duplicate_regs 1
-mux_style Auto
-max_fanout 1000
-fsm_encoding_style Auto
-twr_paths 3
-fix_gated_clocks 1
-loop_limit 1950
-use_io_reg auto
-use_io_insertion 1
-resolve_mixed_drivers 0
-sdc "impl1.ldc"
-path "C:/lscc/radiant/1.0/ispfpga/ice40tp/data" "impl1"
-ver "C:/lscc/radiant/1.0/ip/pmi/pmi.v"
-ver "count_attr.v"
-path "."
```

```
-top count
-udb "counter_impl1.udb"
-output_hdl "counter_impl1.vm"
```

Command 2: post synthesis process

```
postsyn -a iCE40UP -p iCE40UP3K -t UWG30 -sp Worst Case -top -
ldc counter_impl1.ldc -keeprtl -w -o counter_impl1.udb
counter_impl1.vm
```

Command 3: Mapper

```
map "counter_impl1_syn.udb" "impl1.pdc" -o "counter_impl1.udb"
```

Command 4: Placer and router

```
par -f "counter_impl1.p2t" "counter_impl1_map.udb"
"counter_impl1.udb"
```

Command 5: Timer

```
timing -sethld -v 10 -u 10 -endpoints 10  -nperend 1 -html -rpt
"counter_impl1_twr.html" "counter_impl1.udb"
```

Command 6: back annotation

```
backanno "counter_impl1.udb" -n Verilog  -o
"counter_impl1_vo.vo"        -w -neg
```

Command 7: bitstream generation

```
bitgen -w "counter_impl1.udb" -f "counter_impl1.t2b"
```

Tcl Command Reference Guide

The Radiant software supports Tcl (Tool Command Language) scripting and provides extended Radiant software Tcl commands that enable a batch capability for running tools in the Radiant software's graphical interface. The command set and the Tcl Console used to run it affords you the speed, flexibility and power to extend the range of useful tasks that the Radiant software tools are already designed to perform.

In addition to describing how to run the Radiant software's Tcl Console, this guide provides you with a reference for Tcl command line usage and syntax for all Radiant software point tools within the graphical user interface so that you can create command scripts, modify commands, or troubleshoot existing scripts.

About the Radiant software Tcl Scripting Environment The Radiant software development software features a powerful script language system. The user interface incorporates a complete Tcl command interpreter. The command interpreter is enhanced further with additional Radiant software-specific support commands. The combination of fundamental Tcl along with the commands specialized for use with the Radiant software allow the entire Radiant software development environment to be manipulated.

Using the command line tools permits you to do the following:

- Develop a repeatable design environment and design flow that eliminates setup errors that are common in GUI design flows
- Create test and verification scripts that allow designs to be checked for correct implementation
- Run jobs on demand automatically without user interaction

The Radiant software command interpreter provides an environment for managing your designs that are more abstract and easier to work with than using the core Radiant software engines. The Radiant software command interpreter does not prevent use of the underlying transformation tools. You

can use either the TCL commands described in this section or you can use the core engines described in the “Command Line Reference Guide” on page 95.

Additional References If you are unfamiliar with the Tcl language you can get help by visiting the Tcl/tk web site at http://www.tcl.tk. If you already know how to use Tcl, see the Tcl Manual supplied with this software. For information on command line syntax for running core tools that appear as Radiant software processes, such as synthesis, map, par, backanno, and timing, see the “Command Line Reference Guide” on page 95.

See Also ▶“Running the Tcl Console” on page 148

- “Accessing Command Help in the Tcl Console” on page 150
- “Radiant Software Tool Tcl Command Syntax” on page 154
- “Creating and Running Custom Tcl Scripts” on page 150
- “Accessing Command Help in the Tcl Console” on page 150
- Tcl Manual (Windows only)
- Tcl Manual (Linux only)

Running the Tcl Console

The Radiant software TCL Console environment is made available for your use in multiple different ways. In order to take full advantage of the FPGA development process afforded by the Radiant software you must gain access to the Radiant Tcl Console user interface.

On Windows In Windows 7 you can interact with the Tcl Console by any one of the following methods:

- To launch the Radiant software GUI from the Windows Start menu, choose **Start > All Programs > Lattice Radiant Software > Radiant Software**.

  After the the Radiant software loads you can click on the **TCL Console** tab. With the **TCL Console** tab active, you are able to start entering standard syntax TCL commands or the Radiant software specific support commands.

- To launch the **TCL Console** independently from the Radiant software GUI from the Windows Start menu choose **Start > All Programs > Lattice Radiant Software > Accessories > TCL Console**.

  A Windows command interpreter will be launched that automatically runs the **TCL Console**.

- To run the interpreter from the command line, type the following:

  `c:/lscc/radiant/<version_number>/bin/nt64/pnmainc`

  The Radiant **TCL Console** is now available to run.

- To run the interpreter from a Windows 7 PowerShell from the Windows Start menu choose **Start > All Programs > Accessories > Windows PowerShell > Windows PowerShell (x86)**.

  A PowerShell interpreter window will open. At the command line prompt type the following:

  `c:/lscc/radiant/<version_number>/bin/nt64/pnmainc`

  The Radiant **TCL Console** is now available to run.

Note

The arrangement and location of each of the programs in the Windows Start menu will differ depending on the version of Windows you are running.

On Linux In Linux operating systems you can interact with the Tcl Console by one of the following methods:

- To launch the Radiant software GUI from the command line, type the following:

  `/usr/<user_name>/radiant/<version_number>/bin/lin64/radiant`

  The path provided assumes the default installation directory and that the Radiant software is installed. After the Radiant software loads you can click on the **TCL Console** tab. With the **TCL Console** tab active, you are able to start entering standard syntax TCL commands or the Radiant software specific support commands.

- To launch the **TCL Console** independently from the Radiant software GUI from the command line, type the following:

  `/usr/<user_name>/Radiant/<version_number>/bin/lin64/radiantc`

  The path provided assumes the default installation directory and that the Radiant software is installed, and that you have followed the Radiant software for Linux installation procedures. The Radiant **TCL Console** is now ready to accept your input.

  The advantage of running the **TCL Console** from an independent command interpreter is the ability to directly pass the script you want to run to the Tcl interpreter. Another advantage is that you have full control over the Tk graphical environment, which allows you to create you own user interfaces.

See Also ▶"Running the Tcl Console" on page 148

- "Radiant Software Tool Tcl Command Syntax" on page 154
- "Creating and Running Custom Tcl Scripts" on page 150
- "Accessing Command Help in the Tcl Console" on page 150
- Tcl Manual (Windows only)
- Tcl Manual (Linux only)

Accessing Command Help in the Tcl Console

You can access command syntax help for all of the tools in the Tcl Console.

To access command syntax help in the Tcl Console:

1. In the prompt, type **help <tool_name>*** and press **Enter** as shown below:

   ```
   help prj*
   ```

   A list of valid command options appears in the Tcl Console.

2. In the Tcl Console, type the name of the command or function for more details on syntax and usage. For the prj tool, for example, type and enter the following:

   ```
   prj_open
   ```

   A list of valid arguments for that function appears.

Note

Although you can run the Radiant software's core tools such as synthesis, postsyn, map, par, and timing from the Tcl Console, the syntax for accessing help is different. For proper usage and syntax for accessing help for core tools, see the "Command Line Reference Guide" on page 95.

See Also "Running the Tcl Console" on page 148

- "Radiant Software Tool Tcl Command Syntax" on page 154
- "Creating and Running Custom Tcl Scripts" on page 150
- "Running Tcl Scripts When Launching the Radiant Software" on page 153
- Tcl Manual (Windows only)
- Tcl Manual (Linux only)

Creating and Running Custom Tcl Scripts

This topic describes how to easily create Tcl scripts using the Radiant software's user interface and manual methods. FPGA design using Tcl scripts provides some distinct advantages over using the graphical user interface's lists, views and menu commands. For example, Tcl scripts allow you to do the following:

- Set the tool environment to exactly the same state for every design run. This eliminates human errors caused by forgetting to manually set a critical build parameter from a drop-down menu.
- Manipulate intermediate files automatically, and consistently on every run. For example, .vm file errors can be corrected prior to performing additional netlist transformation operations.
- Run your script automatically by using job control software. This gives you the flexibility to run jobs at any time of day or night, taking advantage of idle cycles on your corporate computer system.

Creating Tcl Scripts There are a couple of different methods you can use to create the Radiant software Tcl scripts. This section will discuss each one and provide step-by-step instructions for you to get started Tcl scripting repetitious Radiant software commands or entire workflows.

One method you have available is to use your favorite text editor to enter a sequence of the Radiant software Tcl commands. The syntax of each the Radiant software Tcl commands is available in later topics in this portion of the online help. This method should only be used by very experienced Radiant software Tcl command line users.

The preferred method is to let the Radiant software GUI assist you in getting the correct syntax for each Tcl command. When you interact with the Radiant software user interface each time you launch a *scriptable* process and the corresponding Radiant software Tcl command is echoed to the Tcl Console. This makes it much simpler to get the correct command line syntax for each Radiant software command. Once you have the fundamental commands executed in the correct order, you can then add additional Tcl code to perform error checking, or customization steps.

To create a Tcl command script in the Radiant software:

1. Start the Radiant software design software and close any project that may be open.
2. In the Tcl Console execute the custom **reset** command. This clears the Tcl Console command history.
3. Use the Radiant software graphical user interface to start capturing the basic command sequence. The Tcl Console echos the commands in its window. Start by opening the project for which you wish to create the TCL script. Then click on the processes in the Process bar to run them. For example, run these processes in their chronological order in the design flow:
   - Synthesize Design
   - Map Design
   - Place & Route Design
   - Export Files
4. In the Tcl Console window enter the command,

   ```
   save_script <filename.ext>
   ```

   The <filename.ext> is any file identifier that has no spaces and contains no special characters except underscores. For example, **myscript.tcl** or **design_flow_1.tcl** are acceptable save_script values, but **my$script** or **my script** are invalid. The **<filename.ext>** entry can be preceded with a absolute or relative path. Use the **"/"** (i.e. forward slash) character to delimit the path elements. If the path is not specified explicitly the script is saved in the current working directory. The current working directory can be determined by using the TCL *pwd* command.
5. You can now use your favorite text editor to make any changes to the script you feel are necessary. Start your text editor, navigate to the

directory the *save_script* command saved the base script, and open the file.

Note

In most all cases, you will have to clean up the script you saved and remove any invalid arguments or any commands that cannot be performed in the Radiant software environment due to some conflict or exception. You will likely have to revisit this step later if after running your script you experience any run errors due to syntax errors or technology exceptions.

Sample Radiant software Tcl Script The following the Radiant software Tcl script shows a very simple script that opens a project, runs the entire design flow through the Place & Route process, then closes the project. A typical script will contain more tasks and will check for failure conditions. Use this simple example as a general guideline.

Figure 83: Simple Radiant software Script

```
prj_archive -dir "C:/my_radiant/counter" -extract "C:/lscc/
radiant/1.1/examples/counter.zip"
prj_run_par
prj_close
```

Running Tcl Scripts The Radiant software TCL scripts are run exclusively from the Radiant TCL Console. You can use either the TCL Console integrated into the Radiant software UI, or by launching the stand-alone TCL Console.

To run a Tcl script in the Radiant software:

1. Launch the Radiant software GUI, or the stand-alone TCL Console.

   Open the Radiant software but do not open your project. If your project is open, choose **File > Close Project**.

2. If you are using the Radiant software main window, click the small arrow pane switch in the bottom of the Radiant software main window, and then click on the **Tcl Console tab** in the Output area at the bottom to open the console.

3. Use the TCL *source* command to load and run your TCL script. The *source* command requires, as it's only argument, the filename of the script you want to load and run. Prefix the script file name with any required relative or absolute path information. To run the example script shown in the previous section use:

   ```
   source C:/lscc/radiant/<version_number>/examples/counter/
   myscript2.tcl
   ```

   As long as there are no syntax errors or invalid arguments, the script will open the project, synthesize, map, and place-and-route the design. Once the design finishes it closes the project. If there are errors in the script, you will see the errors in red in the Tcl Console after you attempt to run it. Go back to your script and correct the errors that prevented the script from running.

See Also ▶"Running the Tcl Console" on page 148

- "Radiant Software Tool Tcl Command Syntax" on page 154
- "Running Tcl Scripts When Launching the Radiant Software" on page 153
- Tcl Manual (Windows only)
- Tcl Manual (Linux only)

Running Tcl Scripts When Launching the Radiant Software

This topic describes how launch the Radiant software and automatically run Tcl scripts using a command line shell or the stand-alone Tcl console. Your Tcl script can be standard Tcl commands as well as the Radiant software-specific Tcl commands.

Refer to "Creating and Running Custom Tcl Scripts" on page 150 for more information on creating custom Tcl scripts.

To launch the Radiant software and run a Tcl script from a command line shell or the stand-alone Tcl console:

- Enter the following command:

  On Windows:

  ```
  pnmain.exe -t <tcl_path_file>
  ```

  On Linux:

  ```
  radiant -t <tcl_path_file>
  ```

Sample Radiant software Tcl Script The following Radiant software Tcl script shows a very simple script, running in Windows, that opens a project and runs the design flow through the MAP process. Use this simple example as a general guideline.

Figure 84: Simple Radiant Software Script

```
prj_open C:/test/iobasic_radiant/io1.rdf
prj_run_map
```

The above example is saved in Windows as the file mytcl.tcl in the directory C:/test. By running the following command from either a DOS shell or the Tcl console in Windows, the Radiant software GUI starts, the project io1.rdf opens, and the MAP process automatically runs.

```
pnmain.exe -t c:/test/mytcl.tcl
```

See Also "Running the Tcl Console" on page 148

- "Radiant Software Tool Tcl Command Syntax" on page 154
- "Creating and Running Custom Tcl Scripts" on page 150
- Tcl Manual (Windows only)
- Tcl Manual (Linux only)

Radiant Software Tool Tcl Command Syntax

This part of the Tcl Command Reference Guide introduces the syntax of each of the Radiant software tools and provides you with examples to help you construct your own commands and scripts.

The Radiant software tries to make it easy to develop TCL scripts by mirroring the correct command syntax in the Tcl Console based on the actions performed by you in the GUI. This process works well for most designs, but there are times when a greater degree of control is required. More control over the build process is made available through additional command line switches. The additional switches may not be invoked by actions taken by you when using the GUI. This section provides additional information about all of the Tcl commands implemented in the Radiant software.

The Tcl Commands are broken into major categories. The major categories are:

- Radiant Software Tcl Console Commands
- Radiant Software Timing Constraints Tcl Commands
- Radiant Software Physical Constraints Tcl Commands
- Radiant Software Project Tcl Commands
- Reveal Inserter Tcl Commands
- Reveal Analyzer Tcl Commands
- Power Calculator Tcl Commands
- Programmer Tcl Commands
- Engineering Change Order Tcl Commands

Radiant Software Tcl Console Commands

The Radiant software Tcl Console provides a small number of commands that allow you to perform some basic actions upon the Tcl Console Pane. The Radiant software Tcl Console commands differ from the other Tcl commands

provided in the Radiant software. This dtc program's general Tcl Console commands do not use the *dtc_* prefix in the command syntax as is the convention with other tools in the Radiant software.

Note

TCL Command Log is always listed after the project is closed. You can find it in the Reports section under Misc Report > TCL Command Log.

The following table provides a listing of all valid Radiant software Tcl Console-related commands.

Table 19: Radiant Software Tcl Console Commands

Command	Arguments	Description
clear	N/A	The *clear* command erases anything present in the Tcl Console pane, and prints the current *prompt* character in the upper left corner of the Tcl Console pane without erasing the command history.
history	N/A	The *history* command lists the command history in the Tcl Console that you executed in the current session. Every command entered into the Tcl Console, either by the GUI, or by direct entry in the Tcl Console, is recorded so that it can be recalled at any time. The command history list is cleared when a project is *opened* or when the Tcl Console *reset* command is executed.
reset	N/A	The *reset* command clears anything present in the Tcl Console pane, and erases all entries in the command line history. **It's only used in GUI Tcl console and not supported in stand-alone Tcl console.

Table 19: Radiant Software Tcl Console Commands

Command	Arguments	Description
save_script	<filename.ext>	Saves the contents of the command line history memory buffer into the script file specified. The script is, by default, stored into the current working directory. File paths using forward slashes used with an identifier are valid if using an absolute file path to an existing script folder. **It's only used in GUI Tcl console and not supported in stand-alone Tcl console.
set_prompt	<new_character>	The default prompt character in the Tcl Console is the "greater than" symbol or angle bracket (i.e., >). You can change this prompt character to some other special character such as a dollar sign ($) or number symbol (#) if you prefer. **It's only used in GUI Tcl console and not supported in stand-alone Tcl console.

Radiant Software Tcl Console Command Examples This section illustrates and describes a few samples of Radiant Tcl Console commands.

Example 1 To save a script, you simply use the **save_script** command in the Tcl Console window with a name or file path/name argument. In the first example command line, the file path is absolute, that is, it includes the entire path. Here you are saving "myscript.tcl" to the existing current working directory. The second example creates the same "myscript.tcl" file in the current working directory.

```
save_script C:/lscc/radiant/myproject/scripts/myscript.tcl

save_script myscript.tcl
```

See "Creating and Running Custom Tcl Scripts" on page 150 for details on how to save and run scripts in the Radiant software.

Example 2 The following **set_prompt** command reassigns the prompt symbol on the command line as a dollar sign ($). The default is an angle bracket or "greater than" sign (>).

```
set_prompt $
```

Example 3 The following **history** command will print all of the command history that was recorded in the current Tcl Console session.

```
history
```

Radiant Software Timing Constraints Tcl Commands

The following table provides a listing of all valid Radiant software Timing Constraints Tcl commands.

Table 20: Radiant Software Timing Constraints Tcl Commands

Command	Arguments	Description
create_clock	**create_clock** -period <period_value> [-name <clock_name>] [-waveform <edge_list>] [<port_list|pin_list|net_list>]	Create a named or virtual clock.
create_generated_clock	**create_generated_clock** [-name <clock_name>] -source <master_pin>[-edges <edge_list>] [-divide_by <factor>] [-multiply_by <factor>] [-duty_cycle <percent>] [-invert] [-add] [<pin_list|net_list|port_list>]	Create a generated clock object.
ldc_define_attribute	l**dc_define_attribute** -attr <attr_type> -value <attr_value> -object_type <object type> -object <object> [-disable] [-comment <comment>]	Set LSE synthesis attributes for given objects
set_clock_groups	**set_clock_groups** -group <clock_list> <-logically_exclusive | -physically_exclusive | -asynchronous>	Set clock groups.
set_clock_latency	**set_clock_latency** [-rise] [-fall] [-early | -late] <source> <latency> <object_list>	Defines a clock's source or network latency
set_clock_uncertainty	**set_clock_uncertainty** [-setup] [-hold] [-from <clock>] [-to <clock>] <uncertainty> [<clock_list>]	Set clock uncertainty.

Table 20: Radiant Software Timing Constraints Tcl Commands

Command	Arguments	Description
set_false_path	**set_false_path** [-from <port_list|pin_list|instance_list|net_list|clock_list>] [-to <port_list|pin_list|instance_list|net_list|clock_list>] [-through <port_list|pin_list|instance_list|net_list>] [-rise_from <clock_list>] [-rise_to <clock_list>] [-fall_from <clock_list>] [-fall_to <clock_list>] [-comment string]	Define false path
set_input_delay	**set_input_delay** -clock <clock_name> [-clock_fall] [-max] [-min] [-add_delay] <delay_value> <port_list>	Set input delay on ports
set_load	**set_load** <capacitance> <objects>	Commands to set capacitance on ports
set_max_delay	**set_max_delay** [-from <port_list|pin_list|instance_list|net_list|clock_list>] [-to <port_list|pin_list|instance_list|net_list|clock_list>] [-through <port_list|pin_list|instance_list|net_list>] [-rise_from <clock_list>] [-rise_to <clock_list>] [-fall_from <clock_list>] [-fall_to <clock_list>] <delay_value> [-comment string]	Specify maximum delay for timing paths
set_min_delay	**set_min_delay** [-from <port_list|pin_list|instance_list|net_list|clock_list>] [-to <port_list|pin_list|instance_list|net_list|clock_list>] [-through <port_list|pin_list|instance_list|net_list>] [-rise_from <clock_list>] [-rise_to <clock_list>] [-fall_from <clock_list>] [-fall_to <clock_list>] <delay_value>	Specify minimum delay for timing paths

Table 20: Radiant Software Timing Constraints Tcl Commands

Command	Arguments	Description
set_multicycle_path	**set_multicycle_path** [-from <port_list|pin_list|instance_list|net_list|clock_list>] [-to <port_list|pin_list|instance_list|net_list|clock_list>] [-through <port_list|pin_list|instance_list|net_list>] [-rise_from <clock_list>] [-rise_to <clock_list>] [-fall_from <clock_list>] [-fall_to <clock_list>] [-setup|-hold] [-start|-end] <path_multiplier>	Define multicycle path
set_output_delay	**set_output_delay** -clock <clock_name> [-clock_fall] [-max] [-min] [-add_delay] <delay_value> <port_list>	Set output delay on ports

Radiant Software Physical Constraints Tcl Commands

The following table provides a listing of all valid Radiant software Physical Constraints Tcl commands

Table 21: Radiant Software Physical Constraints Tcl Commands

Command	Arguments	Description
ldc_create_group	**ldc_create_group** -name <group_name> [-bbox {height width}] <objects>	Defines a single identifier that refers to a group of objects
ldc_create_region	**ldc_create_region** -name <region_name> -site <site> -width <width> -height <height>	Define a rectangular area
ldc_create_vref	**ldc_create_vref** -name <vref_name> -site <site_name>	Define a voltage reference
ldc_prohibit	**ldc_prohibit** -site <site> -region <region>	Prohibits the use of a site or all sites inside a region
ldc_set_attribute	**ldc_set_attribute** <key-value list> [objects]	Set object attributes

Table 21: Radiant Software Physical Constraints Tcl Commands

Command	Arguments	Description
ldc_define_global_attribute	l**dc_define_global_attribute** -attr <attr_type> -value <attr_value> [-disable] [-comment <comment>]	Set LSE synthesis global attributes
ldc_define_attribute	l**dc_define_attribute** -attr <attr_type> -value <attr_value> -object_type <object type> -object <object> [-disable] [-comment <comment>]	Set LSE synthesis attributes for given objects
ldc_set_location	**ldc_set_location** [-site <site_name>] [-bank <bank_num>] [-region <region_name>] <object>	Set object location
ldc_set_port	**ldc_set_port** [-iobuf [-vref <vref_name>]]|[-sso] <key-value list> <ports>	Set port constraint attributes
ldc_set_sysconfig	**ldc_set_sysconfig** <key-value list>	Set sysconfig attributes
ldc_set_vcc	**ldc_set_vcc** [-bank bank|-core] [-derate derate] [voltage]	Sets the voltage and/or derate for the bank or core

Radiant Software Physical Constraints Tcl Commands Examples This section illustrates and describes a few samples of Radiant Physical Constraints Tcl commands.

Example 1 TThe following **ldc_create_group** command creates a sample group with 3 instances, and places all instances within the group to a 2x2 bbox.

```
ldc_create_group -name sample_group -bbox {2 2} [get_cells
{i16_1_lut i18_2_lut i25_3_lut }]
```

Example 2 The following **ldc_set_location** command places the port clk to pin E7.

```
ldc_set_location -site {E7} [get_ports clk]
```

Example 3 The following **ldc_set_port** command sets IO_TYPE, DRIVE, SLEWRATE attributes of the port rst.

```
ldc_set_port -iobuf {IO_TYPE=LVCMOS33 DRIVE=8 SLEWRATE=FAST}
[get_ports rst]
```

Radiant Software Project Tcl Commands

The Radiant software Project Tcl Commands allow you to control the contents and settings applied to the tools, and source associated with your design. Projects can be opened, closed, and configured to a consistent state using the commands described in this section.

Radiant Software Project Tcl Command Descriptions The following table provides a listing of all valid Radiant software project-related Tcl command options and describes option functionality.

Table 22: Radiant Software Project Tcl Commands

Command	Function (Argument)	Description
prj_create	**prj_create** -name <project name> [-dev <device name>] [-performance <performance grade>] [-impl <initial implementation name>] [impl_dir <initial implementation directory>] [-synthesis <synthesis tool name>]	Creates a new project inside the current working directory. The *new* command can only be used when no other project is currently open. The -name <project name> argument specifies the name of the project. This creates a <project name>.rdf file in the current working directory. The -impl <initial implementation name> argument specifies the active implementation when the project is created. If this left unspecified a default implementation called “Implemenation0” is created. The -dev <device name> argument specifies the FPGA family, density, footprint, performance grade, and temperature grade to generate designs for. Use the Lattice OPN (Ordering Part Number) for the <device name> argument. The -performance <performance grade> argument specifies the device performance grade explicitly. For iCE40UP device, performance grade can’t be inferred from the device part name such as iCE40UP3K-UWG30ITR. If no performance grade specified, default performance value is used. The -impl_dir <initial implementation directory> argument defines the directory where temporary files are stored. If this is not specified the current working directory is used.
prj_close	**prj_close**	Exits the current project. Any unsaved changes are discarded.

Table 22: Radiant Software Project Tcl Commands

Command	Function (Argument)	Description
prj_open	**prj_open** <projectfile.rdf>	Opens the specified project in the software environment.
prj_save	**prj_save** [projectfile.rdf]	Updates the project with all changes made during the current session and the project file is saved.
prj_saveas	**prj_saveas** -name <new project name> -dir <new project directory> [-copy_gen]	Save the current project as a new project with specified name and directory.
prj_set_opt	**prj_set_opt** : List all the options in the current project **prj_set_opt** <option name> [option value list] : List or set the option value **prj_set_opt** -append <option name> <option value> : Append a value to the specified option value **prj_set_opt** -rem <option name>... : Remove the options of the current project	List, set or remove a project option.
prj_archive	**prj_archive** [-includeAll] <archive_file> : Archive the current project into the archive_file **prj_archive** -extract -dir <destination directory> <archive_file> : Extract the archive file and load the project	Archive the current project.
prj_set_device	**prj_set_device** [-family <family name>] [-device <device name>] [-package <package name>] [-performance <performance grade>] [-operation <operation>] [-part <part name>] : Change the device to the specified family, device, package, performance, operation, part	Set the device.

Table 22: Radiant Software Project Tcl Commands

Command	Function (Argument)	Description
prj_add_source	**prj_add_source** [-impl <implement name>] [-simulate_only|-synthesis_only] [-include <path list for Verilog include search path>] [-work <VHDL lib name>] [[-opt <name=value>] ...] [-exclude] <src file>...	Adds a VHDL source file to the specified or active implementation. The syntax used for the Add function depends upon the source file's implementation language. [-work <VHDL lib name>]: Assigns the source code to the specified library name space. [-impl <implementation name>]: This switch is used to add a source file to a Radiant software implementation. If this switch is not specified the source file is added to the active implementation. [-opt name=value]: The -opt argument allows you to set a custom, user-defined option. See Example 7 for guidelines and usage. <sre file>...: One or more VHDL source files to add to the specified implementation.
prj_enable_source	**prj_enable_source** [-impl <implement name>] <src file> ...	Enables the excluded design sources from the current project, that is, it will activate a source file for synthesis, to be used as a constraint, or for Reveal debugging.
prj_disable_source	**prj_disable_source** [-impl <implement name>] <src file> ...	Disables the excluded design sources from the current project, that is, it will activate a source file for synthesis, to be used as a constraint or for Reveal debugging.
prj_remove_source	**prj_remove_source** [-impl <implement name>] -all :Remove all the design sources in project **prj_remove_source** [-impl <implement name>] <src file>	Deletes the specified source files from the specified implementation. If an implementation is not listed explicitly the source files are removed from the active implementation. The source files are not removed from the file system, they are only removed from consideration in the specified implementation.

Table 22: Radiant Software Project Tcl Commands

Command	Function (Argument)	Description
prj_set_source_opt	**prj_set_source_opt** -src <source name> [-impl <implement name>] : List all the options in the specified source **prj_set_source_opt** -src <source name> [-impl <implement name>] <option name> [option value list] : List or set the source's option value **prj_set_source_opt** -src <source name> [-impl <implement name>] -append <option name> <option value> : Append a value to the specified option value **prj_set_source_opt** -src <source name> [-impl <implement name>] -rem <option name>... : Remove the options of the source	List, set or remove a source option.
prj_create_impl	**prj_create_impl** <new impl name> [-dir <implementation directory>] [-strategy <default strategy name>] [-synthesis <synthesis tool name>]	Create a new implementation in the current project with '<new impl name>'. The new implementation will use the current active implementation's strategy as the default strategy if no valid strategy is set.
prj_remove_impl	**prj_remove_impl** <implement name>	Delete the specified implementation in the current project with '< impl name>'.

Table 22: Radiant Software Project Tcl Commands

Command	Function (Argument)	Description
prj_set_impl_opt	**prj_set_impl_opt** [-impl <implement name>] : List all the options in the specified implementation **prj_set_impl_opt** [-impl <implement name>] <option name> [option value list] : List or set the implementation's option value **prj_set_impl_opt** [-impl <implement name>] -append <option name> <option value> : Append a value to the specified option value **prj_set_impl_opt** [-impl <implement name>] -rem <option name>... : Remove the the options in the implementation	Allows you to add, list, or remove implementation options with the name <implement name> in the specified or active implementation of the current project. If the -rem option is used, the following option names appearing after it will be removed. If no argument is used (i.e., “prj_impl option”), the default is to list all implementation options. If only the <option name> argument is used (i.e., “prj_impl option <option name>), then the value of that option in the project will be returned. The command will set the option value to the option specified by <option name>. If the <option value> is empty then the option will be removed and ignored (e.g., prj_impl option -rem). The -run_flow argument allows you to switch from the normal mode to an “initial” incremental flow mode and “incremental” which is the mode you should be in after an intial design run during the incremental design flow. With no value parameters specified, -run_flow will return the current mode setting.
prj_set_prescript	**prj_set_prescript** [-impl <implement name>] <milestone name> <script_file> : milestone name can be ‘syn’, ‘map’, ‘par’, ‘export’	List or set user Tcl script before running milestone.
prj_set_postscript	**prj_set_postscript** [-impl <implement name>] <milestone name> <script_file> : milestone name can be ‘syn’, ‘map’, ‘par’, ‘export’	List or set user Tcl script after running milestone.
prj_activate_impl	**prj_activate_impl** <implement name>	Activates the implementation with the name <implement name>.
prj_clean_impl	**prj_clean_impl** [-impl <implement name>]	Clean up the implementation result files in the current project.
prj_clone_impl	**prj_clone_impl** <new impl name> [-dir <new impl directory>] [-copyRef] [-impl <original impl name>]	Clone an existing implementation.
prj_run_synthesis	**prj_run_synthesis**	Run synthesis process.

Table 22: Radiant Software Project Tcl Commands

Command	Function (Argument)	Description
prj_run_map	**prj_run_map**	Run map process.
prj_run_par	**prj_run_par**	Run par process.
prj_run_bitstream	**prj_run_bitstream**	Run bitstream process.
prj_create_strategy	**prj_create_strategy** -name <new strategy name> -file <strategy file name>	Create a new strategy with default setting.
prj_remove_strategy	**prj_remove_strategy** <strategy name>	Deletes an existing strategy.
prj_copy_strategy	**prj_copy_strategy** -from <source strategy name> -name <new strategy name> -file <strategy file name>	Copies an existing strategy and saves it to a newly created strategy file.
prj_import_strategy	**prj_import_strategy** -name <new strategy name> -file <strategy file name>	Import an existing strategy file.
prj_set_strategy	**prj_set_strategy** [-impl <implementation name>] <strategy name>	Associate the strategy with the specified implementation.
prj_list_strategy	**prj_list_strategy** [-strategy <strategy name>] <pattern>	List value to a strategy item.

Radiant Software Project Tcl Command Examples This section illustrates and describes a few samples of Radiant software Project Tcl commands.

Example 1 To create a new project, your command may appear something like the following which shows the creation of a ThunderPlus device.

```
prj_create -name "m" -impl "m" -dev iCE40UP3K-UWG30ITR
```

Example 2 To save a project and give it a certain name (save as), use the project save command as shown below:

```
prj_save "my_project"
```

To simply save the current project just use the save function with no values:

```
prj_save
```

Example 3 To open an existing project, the command syntax would appear with the absolute file path on your system as shown in the following example:

```
prj_open "C:/projects/radiant/adder/my_project.rdf"
```

Example 4 To add a source file, in this case a source LDC file, use the prj_src add command as shown below and specify the complete file path:

```
prj_add_source "C:/my_project/radiant/counter/counter.ldc"
```

Example 5 The following examples below shows the prj_run command being used:

```
prj_run_par
```

In this final example, synthesis is run.

```
prj_run_synthesis
```

Example 6 To copy another project strategy that is already established in another Radiant software project from your console, use the prj_copy_strategy copy command as shown below and specify the new strategy name and the strategy file name.

```
prj_copy_strategy -from source_strategy -name new_strategy -
file strategy.stg
```

Example 7 The prj_add_source command allows you to set a custom, user-defined option. This -opt argument value, however, cannot conflict with existing options already in the system, that is, its identifier must differ from system commands such as "include" and "lib" for example. In addition, a user-defined option may not affect the internal flow but can be queried for any usage in a user's script to arrange their design and sources. All user-defined options can be written to the Radiant software project RDF file.

In the example below, the -opt argument is used as a qualifier to make a distinction between to .rvl file test cases.

```
prj_add_source test1.rvl -opt "debug_case=golden_case"
prj_add_source test2.rvl -opt "debug_case=bad_case"
```

Example 8 After you modify your strategy settings in the Radiant software interface the values are saved to the current setting via a Tcl command. For example, a command similar to the following will be called if Synplify frequency and area options are changed.

```
prj_set_strategy_value -strategy strategy1 SYN_Frequency=300
SYN_Area=False
```

Simulation Libraries Compilation Tcl Commands

This section provides Simulation Libraries Compilation extended Tcl command syntax and usage examples.

Simulation Libraries Compilation Tcl Command Descriptions The following table provides a listing of all valid Simulation Libraries Compilation Tcl Command arguments and describes their usage.

Note

Running cmpl_libs may take a long time and may cause the Radiant software to hang.

- It is recommended to run cmpl_libs using the Radiant TCL Console (**Start Menu > Lattice Radiant Software > Accessories > TCL Console**).

or,

- Run cmpl_libs.tcl using the command line console. Refer to "Running cmpl_libs.tcl from the Command Line" on page 103.

Table 23: Simulation Libraries Compilation Tcl Command

Command	Function (Argument)	Description
cmpl_libs	**-sim_path** <sim_path> [-sim_vendor {mentor<default>}] [-device {ice40up|LIFCL|all<default>}] [-target_path <target_path>]	The -sim_path argument specifies the path to the simulation tool executable (binary) folder. This option is mandatory. Currently only Modelsim and Questa simulators are supported. NOTE: If you are a Windows user and prefer the \ notation in the path, you must surround it with {}. And "" or {} will be needed if the path has spaces. NOTE: To execute this command error free, Questasim 10.4e or a later 10.4 version, or Questasim 10.5b or a later version should be used for compilation. The -sim_vendor argument is optional, and intended for future use. It currently supports only Mentor Graphics simulators (Modelsim / Questa). The -device argument specifies the Lattice FPGA device to compile simulation libraries for. This argument is optional, and the default is to compile libraries for all the Lattice FPGA devices. The -target_path argument specifies the target path, where you want the compiled libraries and modelsim.ini file to be located. This argument is optional, and the default target path is the current folder. NOTES: (1) This argument is recommended if you did not change the current folder from the Radiant software startup (binary) folder, or if the current folder is write-protected. (2) If you are a Windows user and prefer the \ notation in the path, you must surround it with {}. And "" or {} will be needed if the path has spaces.

Simulation Libraries Compilation Tcl Command Examples This section illustrates and describes a few examples of Simulation Libraries Compilation Tcl command.

Example 1 The following command will compile all the Lattice FPGA libraries for both Verilog and VHDL simulation, and place them under the folder specified by -target_path. The path to Modelsim is specified by -sim_path.

```
cmpl_libs -sim_path C:/questasim64_10.4e/win64 -target_path c:/
mti_libs
```

Reveal Inserter Tcl Commands

This section provides Reveal Inserter extended Tcl command syntax, command options, and usage examples.

Reveal Inserter Tcl Command Descriptions The following table provides a listing of all valid Reveal Inserter Tcl command options and describes option functionality.

Table 24: Reveal Inserter Tcl Commands

Command	Function (Argument)	Description
rvl_new_project	**rvl_new_project** <rvl file>	Create a new reveal inserter project.
rvl_open_project	**rvl_open_project** <rvl file>	Open a reveal inserter project file.
rvl_save_project	**rvl_save_project** <rvl file>	Save the current reveal inserter project.
rvl_close_project	**rvl_close_project**	Close the current reveal inserter project.
rvl_run_project	**rvl_run_project** [-save] [-saveAs <file>] [-overwrite] [-drc] [-insert_core <core_name>]	▶ "Run inserting debug core task or DRC checking on the current reveal inserter project ▶ -save: Save the project before run command ▶ -saveAs: Save as a different file before run command ▶ -overwrite: Overwrite the existing file if the saved as to file exists already ▶ -drc: Run DRC checking only ▶ -insert_core: Specify the core to be inserted. All cores will be inserted if none is specified"
rvl_add_core	**rvl_add_core** <core name>	Add a new core in current project.
rvl_del_core	**rvl_del_core** <core name>	Remove an existing core from current project.

Table 24: Reveal Inserter Tcl Commands

Command	Function (Argument)	Description
rvl_rename_core	**rvl_rename_core** <core name> <new core name>	Rename an existing core from current project.
rvl_set_core	**rvl_set_core** [core name]	List the default core or select a core as the default core in current project.
rvl_list_core	**rvl_list_core**	List all cores in current project.
rvl_add_serdes	**rvl_add_serdes**	Add the Serdes core into current project.
rvl_del_serdes	**rvl_del_serdes**	Remove the Serdes core from current project.
rvl_set_serdes	**rvl_set_serdes** [clk=<clock name>] [rst=<reset signal, default value is VLO>]	List or set options of Serdes core.
rvl_add_controller	**rvl_add_controller**	Add the Controller Core into current project.
rvl_del_controller	**rvl_del_controller**	Remove the Controller Core from current project.
r**vl_set_controller**	**rvl_set_controller** [-item LED|Switch|Register|PLL1|PLL2|...] [-set_opt {opt_list}] [-set_sig {sig_list}]	List or set options of Controller items in current project You can set opt_list with the following: ▶ Insert=[on|off] for all item ▶ Width=[1..32] for LED and Switch ▶ AddrWidth=[4..16] for Register ▶ DataWidth=[4..32] for Register. sig_list with the following: ▶ SWn=signal where n=1 to Width for Switch. ▶ LEDn=signal where n=1 to Width for LED. ▶ Clock=clk_signal for Register. ▶ Enable=en_signal for Register. ▶ Wr_Rdn=wr_rdn_signal for Register. ▶ Address=addr_bus for Register. ▶ WData=wdata_bus for Register. ▶ RData=rdata_bus for Register.
rvl_add_trace	**rvl_add_trace** [-core <core name>] [-insert_at <position>] <signals list>	Add trace signals in a debug core in current project. You can specify an existing trace signal/bus name or a postion number in a trace bus as the inserting position.
rvl_del_trace	**rvl_del_trace** [-core <core name>] <signals list>	Delete trace signals in a debug core in current project.

Table 24: Reveal Inserter Tcl Commands

Command	Function (Argument)	Description
rvl_rename_trace	**rvl_rename_trace** [-core <core name>] -bus <bus name> <new bus name>	Change the name of a trace bus in a debug core in current project.
rvl_list_trace	**rvl_list_trace** [-core <core name>]	List all trace signals in a debug core in current project.
rvl_move_trace	**rvl_move_trace** [-core <core name>] [-move_to <position>] <signals list>	Move and rearrange the order of trace signals in a debug core in current project. You can specify an existing trace signal/bus name or a postion number in a trace bus as the new position.
rvl_group_trace	**rvl_group_trace** [-core <core name>] -bus <bus name> <signals list>	Group specified trace signals in a debug core in current project into a bus.
rvl_ungroup_trace	**rvl_ungroup_trace** [-core <core name>] <bus name>	Ungroup trace signals in a trace bus in a debug core in current project.
rvl_set_traceoptn	**rvl_set_traceoptn** [-core <core name>] [option=value]	List or set trace options of a debug core in current project. You can set the following option: SampleClk = [signal name].
rvl_set_trigoptn	**rvl_set_trigoptn** [-core <core name>] [option=value]	List or set trigger options of a debug core in current project. You can set the following option: DefaultRadix = [bin|oct|dec|hex] EventCounter = [on|off] CounterValue = [2,4,8,16,...,65536] (depend on FinalCounter is on) TriggerOut = [on|off] OutNetType = [IO|NET|BOTH] (depend on TriggerOut is on) OutNetName = [net name] (depend on TriggerOut is on) OutNetPri = [Active_Low|Active_High] (depend on TriggerOut is on) OutNetMPW = [pulse number] (depend on TriggerOut is on).
rvl_list_tu	**rvl_list_tu** [-core <core name>]	List all trigger units in a debug core in current project.

Table 24: Reveal Inserter Tcl Commands

Command	Function (Argument)	Description
rvl_add_tu	**rvl_add_tu** [-core <core name>] [-radix <bin|oct|dec|hex>] [-name <new TU name>] <TU definition>	Add a new trigger unit to a debug core in current project. TU definition format: "{signal list} Operator Value" Operator must be "==", "!=", ">", ">=", "<", "<=", ".RE."(rising edge), ".FE."(falling edge) and ".SC."(serial compare). A default trigger unit name will be created if it's omitted in command..
rvl_del_tu	**rvl_del_tu** [-core <core name>] <TU name>	Remove an existing core from current project.
rvl_rename_tu	**rvl_rename_tu** [-core <core name>] <old name> <new name>	Rename an existing core in current project.
rvl_set_tu	**rvl_set_tu** [-core <core name>] [-radix <bin|oct|dec|hex>] -name <TU name> [-add_sig <signal list>] [-del_sig <signal list>] [-set_sig <signal list>] [-expr <TU definition>] [-op operator] [-val value]	Set a trigger unit in a debug core in current project. TU definition format: "{signal list} Operator Value" Operator must be "==", "!=", ">", ">=", "<", "<=", ".RE."(rising edge), ".FE."(falling edge) and ".SC."(serial compare)..
rvl_list_te	**rvl_list_te** [-core <core name>]	List all trigger expressions in a debug core in current project.
rvl_add_te	**rvl_add_te** [-core <core name>] [-ram <EBR|Slice>] [-name <new TE name>] [-expression <expression string>] [-max_seq_depth <max depth>] [-max_event_count <max event count>	Add a new trigger expression to a debug core in current project. A default trigger expression name will be created if it's omitted in command.
rvl_del_te	**rvl_del_te** [-core <core name>] <TE name>	Delete an existing trigger expression in a debug core in current project.
rvl_rename_te	**rvl_rename_te** [-core <core name>] <old name> <new name>	Rename an existing trigger expression in a debug core in current project.
rvl_set_te	**rvl_set_te** [-core <core name>] [-ram <EBR|Slice>] [-expression <expression string>] [-max_seq_depth <max depth>] [-max_event_count <max event count>] <TE name>	Change an existing trigger expression in a debug core in current project.

Reveal Inserter Tcl Command Examples This section illustrates and describes a few samples of Reveal Inserter Tcl commands.

Example 1 To create a new Reveal Inserter project with the .rvl file extension in your project directory, use the rvl_project command as shown below using the new option.

```
rvl_new_project my_project.rvl
```

Example 2 The following example shows how to set up TU parameters for Reveal Inserter:

```
rvl_set_tu -name TU -add_sig {count[7:0]} -op == -val C3 -radix
Hex
```

Reveal Analyzer Tcl Commands

This section provides Reveal Analyzer extended Tcl command syntax, command options, and usage examples.

Reveal Analyzer Tcl Command Descriptions The following table provides a listing of all valid Reveal Analyzer Tcl command options and describes option functionality.

Table 25: Reveal Analyzer Tcl Commands

Command	Function (Argument)	Description
rva_new_project	**rva_new_project** <file>	Create a new Reveal Analyzer project.
rva_open_project	**rva_open_project** <file>	Open a Reveal Analyzer project file.
rva_save_project	**rva_save_project** <file>	Save the current Reveal Analyzer project.
rva_close_project	**rva_close_project** <file>	Close the current Reveal Analyzer project.
rva_export_project	**rva_export_project** -vcd <file name> [-module <title>]	Export VCD file. Optional to include a title in the VCD file. By default the title will be “<unknown>”.
	rva_export_project -txt <file name> [-siglist <signal list>]	Export TEXT file. Optional to export selected signal list only. By default all signals are exported.

Table 25: Reveal Analyzer Tcl Commands

Command	Function (Argument)	Description
rva_set_project	**rva_set_project** [-frequency <val] [-period <val>] [-tckdelay <val>] [-cabletype <val>] [-cableport <val>]	No arguments specified will return options. -frequency: sets the frequency value for sample clock in MHz -period: sets a period value for sample clock in ns or ps -tckdelay: sets a TCK clock pin pulse width delay value -cabletype: sets the type of cable. Values are LATTICE|USB|USB2 -cableport: sets the port number as integer >= 0.
rva_run	**rva_run**	Runs until trigger condition to capture data.
rva_stop	**rva_stop**	Stops without capturing data.
rva_manualtrig	**rva_manualtrig**	Manual Trigger to capture data.
rva_get_trace	**rva_get_trace**	Lists all trace signals in a core.
rva_set_core	**rva_set_core** [-name <name>] [-run <on|off>]	No arguments return list of core. -name: Select core. Needed for other actions -run: Turns run option on/off for core.
rva_set_tu	**rva_set_tu** [-name <name>] [-operator {== | != | > | >= | < | <= | "rising edge" | "falling edge"}] [-value <value>] [?radix {bin | oct | dec | hex | <token>}]	No arguments, return list of TU. -name: Select TU. If no options, return options and value for the selected TU. -operator: Sets the comparison operator. Operators are equal to (==), not equal to (!=), greater than (>), greater than or equal to (>=), less than (<), less than or equal to (<=), "rising edge", "falling edge", and serial compare (serial). -value: Sets TU value -radix: Sets TU radix. Options are binary (bin), octal (oct), decimal (dec), hexadecimal (hex), or the name of a token set.
rva_rename_tu	**rva_rename_tu** <name> <new name>	This function renames TU.
rva_set_te	**rva_set_te [**-name <name>] [-expression <expression list>] [-enable <on|off>]	No arguments, return list of TE. -name: Select TE. If no options, return options and value for the selected TE. -expression: Sets TE expression -enable: Enables/disables TE.

Table 25: Reveal Analyzer Tcl Commands

Command	Function (Argument)	Description
rva_rename_te	**rva_rename_te** <name> <new name>	This function renames TE.
rva_set_trigoptn	**rva_set_trigoptn** [-teall <AND|OR>] [-finalcounter <on|off>] [-finalcountervalue <val>] [-samples <val>] [-numtriggers <val>] [-position <pre|center|post|val>]	No arguments specified will return list of options. -teall: Sets AND ALL or OR ALL for all TEs -finalcounter: Turns final trigger counter on/off -finalcountervalue: Sets final trigger counter value -samples: Sets number of samples to capture -numtriggers: Sets number of triggers to capture -position: Sets trigger position to pre-selected or user value.
rva_add_token	**rva_add_token** <tokenset name> <name=value>	Add a token with new name and value in a specific token set.
rva_del_token	**rva_del_token** <tokenset name> <token name>	Delete a specific token in a specific token set.
rva_set_token	**rva_set_token** <tokenset name> <token name> -name <new token name> -value <new token value>	Select specific token in specific token set. -name: Set token name -value: Set token value.
rva_add_tokenset	**rva_add_tokenset** [-tokenset <tokenset name>] [-bits <token bits>] [-token <name=value>]	No arguments, add a token set with default name and bits. -tokenset: Set token set name -bits: Set toke nset bits -token: Add extra tokens.
rva_del_tokenset	**rva_del_tokenset** <tokenset name>	Delete the specific token set.
	rva_del_tokenset -all	Delete all token set.
rva_set_tokenset	r**va_set_tokenset** <tokenset name> -name <new token set name> -bits <new token bits>	Select specific token set -name: Rename a token set -bits: Set number of bits in tokens.
rva_export_tokenset	**rva_export_tokenset** <file name>	Export all token set to a specific file.
rva_import_tokenset	**rva_import_tokenset** <file name>	Import and merge all token set from a specific file.
rva_open_controller	**rva_open_controller**	Open Controller connection to Lattice device before read/write begins.
rva_target_controller	**rva_target_controller**	Set Controller core as target before read/write begins.

Table 25: Reveal Analyzer Tcl Commands

Command	Function (Argument)	Description
rva_close_controller	**rva_close_controller**	Close Controller connection to Lattice device after read/write finished.
rva_read_controller	**rva_read_controller** -addr <addr32>	Read data from 32-bit address in hex.
rva_write_controller	**rva_write_controller** -addr <addr32> -data <data>	Write data to 32-bit address in hex.
rva_run_controller	**rva_run_controller** -read_led|-read_switch|-write_switch <data>|-dump_memfile <mem_file>|-load_memfile <mem_file>|-read_ip <ipname>|-write_ip <ipname>	Run command for Virtual LED, Virtual Switch, User Register, and Hard IP. -read_led: Read data from Virtual LED. -read_switch: Read data from Virtual Switch. -write_switch: Write data to Virtual Switch. -dump_memfile: Dump data from User Register to mem_file. -load_memfile: Load data from mem_file to User Register. -read_ip: Read data from Hard IP register. -write_ip: Write data to Hard IP register.
rva_set_controller	**rva_set_controller** -option <value>	Set the options for Controller core. -cable_type: Set type of cable as USB or USB2. -cable_port: Set logical port of cable as integer. If no arguments specified, then return list of options and values.
rva_export_controller	**rva_export_controller** <rvc_file>	Export Controller options to RVC file.
rva_import_controller	**rva_import_controller** <rvc_file>	Import Controller options from RVC file.

Reveal Analyzer Tcl Command Examples This section illustrates and describes a few samples of Reveal Analyzer Tcl commands.

Example 1 The following command line example shows how to specify a new project that uses a parallel cable port.

```
rva_new_project –rva untitled –rvl "count.rvl" –dev "LFXP2-5E:0x01299043" –port 888 –cable LATTICE
```

Example 2 The following example shows how to set up TU parameters for Reveal Analyzer:

```
rva_set_tu -name TU1 -operator == -value 10110100 -radix bin
```

Power Calculator Tcl Commands

This section provides Power Calculator extended Tcl command syntax, command options, and usage examples.

Power Calculator Tcl Command Descriptions The following table provides a listing of all valid Power Calculator Tcl command options and describes option functionality.

Table 26: Power Calculator Tcl Commands

Command	Function (Argument)	Description
pwc_new_project	**pwc_new_project** <file>	Create a new project.
pwc_open_project	**pwc_open_project** <file>	Open a project file.
pwc_save_project	**pwc_save_project** <file>	Save the current project.
pwc_close_project	**pwc_close_project**	Close the current project.
pwc_set_afpervcd	**pwc_set_afpervcd** <file>	Open vcd file and set frequency and activity factor.
pwc_set_device	**pwc_set_device** -family <family name>	Set family.
	pwc_set_device -device <device name>	Set device.
	pwc_set_device -package <package name>	Set package.
	pwc_set_device -speed <speed name>	Set speed.
	pwc_set_device -operating <operating name>	Set operating.
	pwc_set_device -part <part name>	Set part.
pwc_set_processtype	**pwc_set_processtype** <value>	Set device power process type.
pwc_set_ambienttemp	**pwc_set_ambienttemp** <value>	Set ambient temperature value.
pwc_set_thetaja	**pwc_set_thetaja** <value>	Set user defined theta JA.
pwc_set_freq	**pwc_set_freq** <frequency>	Set default frequency.
	pwc_set_freq -clock <frequency>	Set Clock frequency.
	pwc_set_freq -timing <option> option: min | pref | trace	Set frequency by timing.
pwc_set_af	**pwc_set_af** <value>	Set default activity factor.
pwc_set_estimation	**pwc_set_estimation** <value>	Sets estimated routing option.
pwc_set_supply	**pwc_set_supply** -type <value> -voltage <value> -dpm <value>	Set multiplication factor and voltage of named power supply.

Table 26: Power Calculator Tcl Commands

Command	Function (Argument)	Description
pwc_add_ipblock	**pwc_add_ipblock** -iptype <iptype name>	Add IP Block row.
pwc_set_ipblock	**pwc_set_ipblock** -iptype <iptype name> -matchkeys {<key1> <value1>}+ -setkey <key> <value> : iptypename mapping to PGT section, key mapping to _KEY in pgt session, value is its value	Set IP Block row.
pwc_remove_ipblock	**pwc_remove_ipblock** -iptype <iptype name> -matchkeys {<key1> <value1>}+	Remove IP Block row.
pwc_gen_report	**pwc_gen_report** <file>	Generate text report and write to file.
pwc_gen_htmlreport	**pwc_gen_htmlreport** <file>	Generate HTML report and write to file.

Power Calculator Tcl Command Examples This section illustrates and describes a few samples of Power Calculator Tcl commands.

Example 1 The follow command below creates a PWC project (.pcf) file namced “abc.pcf” from an input UDB file named “abc.UDB”:

```
pwc_new_project abc.pcf -udb abc.udb
```

Example 2 To set the default frequency to, for example, 100 Mhz:

```
pwc_set_freq 100
```

Example 3 The command below saves the current project to a new name:

```
pwc_save_project newname.pcf
```

Example 4 To create an HTML report, you would run a command like the one shown below:

```
pwc_gen_htmlreport c:/abc.html
```

Programmer Tcl Commands

This section provides the Programmer extended Tcl command syntax, command options, and usage examples. The below commands are only supported in standalone Programmer currently.

Programmer Tcl Command Descriptions The following table provides a listing of all valid Programmer Tcl command options and describes option functionality.

Table 27: Programmer Tcl Commands

Command	Function (Argument)	Description
pgr_project	**pgr_project open** <project_file>	The open command will open the specified project file in-memory.
	pgr_project save [<file_path>]	Writes the current project to the specified path. If there is no file path specified then it will overwrite the original file.
	pgr_project close	Closes the current project. If a Programmer GUI is open with the associated project, then the corresponding Programmer GUI will be closed as well.
	pgr_project help	Displays help for the pgr_project command.

Table 27: Programmer Tcl Commands

Command	Function (Argument)	Description
pgr_program	<no_argument>	When pgr_program is run without arguments it will display the current status of the available settings. Note that specifying a key without a value will display the current value. The following keys can be used to modify those settings. Generally, the pgr_program command and its sub-commands allow you to run the equivalent process commands from the TCL Console window in the Radiant software interface. These commands can override connection options that are set in user defaults.
	pgr_program set -cable <LATTICE | USB | USB2>	Sets the cable for downloading.
	pgr_program set -portaddress <0x0378|0x0278|0x03bc|0x0378|0x0278|0x03bc|0x<custom address>> <EzUSB-0|EzUSB-1|EzUSB-2|...|EzUSB-15> <FTUSB-0|FTUSB-1|FTUSB-2|...|FTUSB-15>	Sets the port address for the downloading.
	pgr_program run	Executes the current xcf with the current settings. Note that there may be warnings that are displayed in the TCL Console window. These warnings will be ignored and processing will continue.
	pgr_program help	Displays help for pgr_program command.
pgr_genfile	<no_argument>	Programmer generate files command (not supported for customer use)
	pgr_genfile set -process <svf | vme12>	Sets file type for file generation.
	pgr_genfile set -outfile <file path>	Sets the output file.
	pgr_genfile run	Generates file based on the current xcf and current settings.
	pgr_genfile help	Displays help for pgr_genfile command.

Programmer Tcl Command Examples This section illustrates and describes a few samples of Programmer Tcl commands.

Example 1 The first command below opens a Programmer XCF project file that exists in the system. There can be many programming files associated with one project. In the GUI interface, the boldfaced file in the Radiant software is the active project file.>

```
pgr_project open /home/mdm/config_file/myfile.xcf
```

Example 2 The following command sets programming option using a USB2 cable at port address “FTUSB-1, then using pgr_program run to program”.

```
pgr_program set -cable USB2 -portaddress FTUSB-1
```

Example 3 The following command sets the file generation type for JTAG SVF file, then using pgr_genfile run to generates an output file “mygenfile.svf” in a relative path.

```
pgr_genfile set -process svf -outfile ../genfiles/mygenfile.svf
```

Engineering Change Order Tcl Commands

This section provides Engineering Change Order (ECO) extended Tcl command syntax, command options, and usage examples.

ECO Tcl Command Descriptions The following table provides a listing of all valid ECO Tcl command options and describes option functionality.

Table 28: ECO Tcl Commands

Command	Function (Argument)	Description
eco_save_design	**eco_save_design** [-udb <udb_file>]	Saves an existing design or macro.
eco_config_sysio	**eco_config_sysio** -comp <comp name> {<key=value>}...	Config sysio setting.
eco_config_memory	**eco_config_memory** -mem_id <memory_id> {-init_file <mem_file> -format HEX|BIN|ADDR} | -all_0 | -all_1	Update memory initial value.

ECO Tcl Command Examples This section illustrates and describes a few samples of ECO Tcl commands.

Example 1 The following demonstrates the `sysio` command:

```
eco_config_sysio -comp {data}
{clamp=OFF;diffdrive=NA;diffresistor=OFF;drive=2;glitchfilter=
OFF;hysteresis=NA;opendrain=OFF;pullmode=NONE;slewrate=SLOW;te
rmination=OFF;vref=OFF}
```

Example 2 The following demonstrates the `memory` command:

```
eco_config_mem -mem_id {mem} -init_file {D:/mem/init_hex.mem}
-format HEX
```

Chapter 10

Advanced Topics

This chapter explains advanced concepts, features and operational methods for the Radiant software.

Shared Memory Environment

The Radiant software design environment uses a shared memory architecture. Shared memory allows all internal tool views to access the same image of the design at any point in time. Understanding how shared memory is being used can give you insight into managing the environment for optimum performance, especially when your design is large.

There is one shared database that contains the device, design, and constraint information in system memory.

Generating the hierarchy of your design uses an additional database separate from the primary shared memory database.

External tools referenced from within the Radiant software, such as those for synthesis and simulation, use their own memory in addition to what is used by the Radiant software.

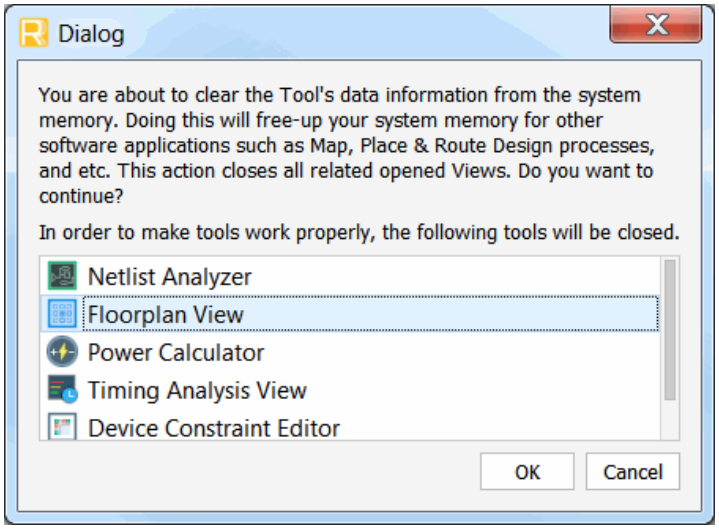
Because it is accessing shared memory, the initial tool view launch will take longer than the launch of subsequent views.

Clear Tool Memory

The “Clear Tool Memory command, available from the Tools menu, clears the device, design, and constraint information from system memory. Clearing the tool memory can speed up memory-intensive processes such as place and route. When your design is very large, it is good practice to clear memory prior to running place and route.

If you have open tool views that will be affected by clearing the tool memory, a confirmation dialog box will open to give you the opportunity to cancel the memory clear.

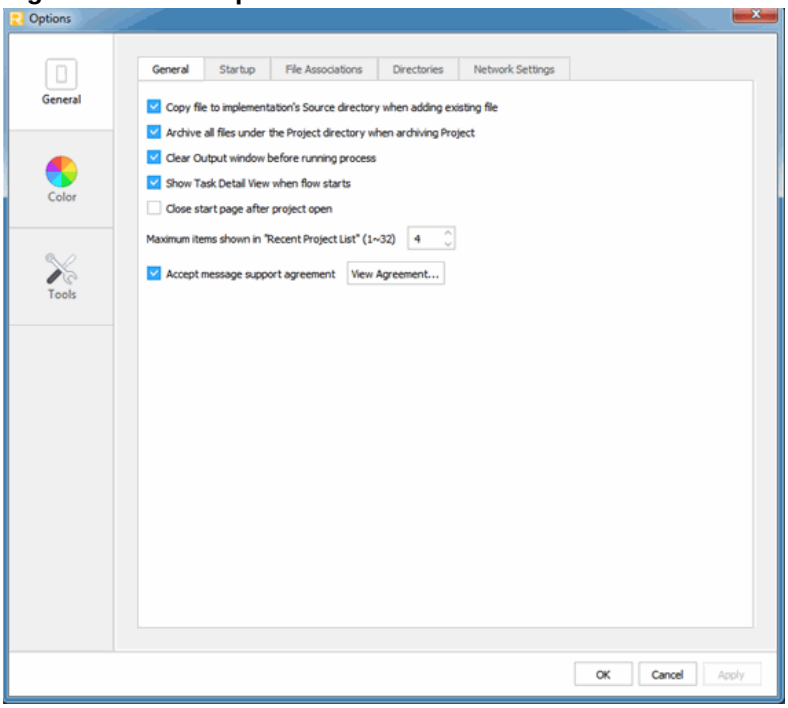
Figure 85: Clear Tool Memory Dialog



Environment and Tool Options

The Radiant software provides many environment control and tool view display options that enable you to customize your settings. Choose **Tools > Options** to access these options.

Figure 86: Tools Options



The Options dialog box is organized into functional folders.

Commonly configured items include:

- General settings -- allows you to set some common options, including.
  - Startup – enables you to configure the default action at startup and also to control the frequency of checking for software updates.
  - File Associations – allows you to set the programs to be associated with different file types based on the file extensions.
  - Directories – Set directory location for Synthesis and Simulation tools.
  - Network Settings – Apply a proxy server and specify a host and port.
- Color settings -- allows you to set colors for such GUI features as fonts and backgrounds for various Radiant software tools. You can also change the Theme color of the Radiant software (Dark or Light) using the Themes drop-down menu.
- Tools settings -- allows you to set options for various Radiant software tools, including the Device Constraint Editor, Netlist Analyzer, Timing Constraint Editor, and Source Editor. You can also set the constraint design rule check (DRC) time to real time, prior to saving or when launching a tool.

Batch Tool Operation

The core tools in the FPGA implementation design flow can all be run in batch mode using command-line tool invocation or scripts. For detailed information, see the *Command Line Reference Guide*, available from the Radiant software Start Page.

Tcl Scripts

The Radiant Extended Tcl language enables you to create customized scripts for tasks that you perform often in the Radiant software. Automating these operations through Tcl scripts not only saves time, but also provides a uniform approach to design. This is especially useful when you try to find an optimal solution using numerous design implementations.

Creating Tcl Scripts from Command History

A good first step in Tcl programming is to create a Tcl script by saving some command history and then modifying it as needed. This allows you to get started by using existing command information.

To create a Tcl command script using command history:

1. In the Tcl Console window, perform a *reset* command so that your script won't contain any of the actions that may have already been executed.

```
reset
```

2. Open the project and perform the commands that you want to save as a script.
3. Optionally, enter the history command in the Tcl Console window to ensure that the commands you wish to save are in the console's memory. In the Tcl Console window type

   ```
   save_script <filename.tcl>
   ```

   The <filename.tcl> can be any file identifier that has no spaces and contains no special characters except underscores. For example, **myscript.tcl** or **design_flow_1.tcl** are acceptable save_script values, but **my$script** or **my script** are invalid. A file name with an extension of .ext will not work.

   The **<filename.ext>** entry can be preceded with an absolute or relative path. Use the **"/"** (i.e. forward slash) symbol to delimit the path elements. If the path is not specified, the script is saved in the current working directory. The current working directory can be determined by using the TCL *pwd* command.
4. Navigate to your script file and use the text editing tool of your choice to make any necessary changes, such as deleting extraneous lines or invalid arguments.

In most cases, you will need to edit the script you saved and take out any invalid arguments or any commands that cannot be performed in the Radiant software environment due to a conflict or exception. You will likely have to revisit this step later if after running your script, you experience any run errors due to syntax errors or technology exceptions.

Creating Tcl Scripts from Scratch

Tcl commands can be written directly into a script file. You can use any text editor, such as Notepad or vi, to create a file and type in the Tcl commands.

Sample Tcl Script

The following Tcl example shows a simple script that opens a project, runs the entire design flow through the Place & Route process, and then closes the project. A typical script would probably contain more steps, but you can use this example as a general guideline.

```
prj_project open "C:/lscc/Radiant/counter/counter.rdf"
prj_run PAR -impl counter -forceAll
prj_project close
save_script c:/lscc/radiant/examples/counter/myscript2.tcl
```

Running Tcl Scripts

You can run scripts from the Radiant software integrated Tcl Console whether your project is opened or not. You can also run scripts from the external Tcl Console prompt window. The following example procedure uses the integrated Tcl Console and the sample Tcl script from the previous section:

To run a Tcl script that opens a project, runs processes and closes the project:

1. Open the Radiant software but do not open your project. If your project is open, choose **File > Close Project**.
2. If you are using the Radiant software main window, click the small arrow pane switch in the bottom of the Radiant software main window, and then click on the **Tcl Console tab** in the Output area at the bottom to open the console.
3. If there are previously issued commands in the console, type **`reset`** in the console command line to refresh your session and clear out all previous commands.

   ```
   reset
   ```

4. Use the TCL *source* command to load and run your TCL script. Since it's the only argument, the *source* command requires the filename of the script you want to load and run. Prefix the script file name with any required relative or absolute path information. To run the example script shown in the previous section use the following:

   ```
   source C:/lscc/radiant/<version_number>/examples/counter/
   myscript2.tcl
   ```

5. As long as there are no syntax errors or invalid arguments, the script will open the project, synthesize, map, and place-and-route the design. Once the design finishes it closes the project. If there are errors in the script, you will see the errors in red in the Tcl Console after you attempt to run it. Go back to your script and correct the errors that prevented the script from running.

Project Archiving

A Radiant software project archive is a single compressed file (.zip) of your entire project. The project archive can contain all of the files in your project directory, or it can be limited to source-related files. When you use the **File > Archive Project** command, the dialog box provides the option to "Archive all files under the Project directory." When you select this option, the entire project is archived. When you clear this option, only the project's source-related files, including strategies, are archived. Many of these source-related files must be archived in order to achieve the same bitstream results for a fully implemented design.

Whichever archiving method you select, if your project contains source files stored outside the project folder, the remote files will be compressed under the remote_files subdirectory in the archive; for example:

```
<project_name>/remote_files/sources
<project_name>remote_files/strategies
```

When unarchiving, you must manually move the archived remote files to the original locations or the project will not work.

File Descriptions

This section provides a list of the file types used in the Radiant software, including those generated during design implementation. The Archive column indicates the files that must be archived in order to achieve the same bitstream results.

Table 29: Source Files

File Type	Definition	Function	Archive?
.fdc	FPGA Design Constraint file	Used for specifying design constraints for Synplify-Pro synthesis tool.	✓
.ipk	Radiant Software IP Package file.	Package file for Radiant software Soft IP.	
.ipx	Manifest file generated by IP Catalog.	Enables changes to be made to a module or IP Catalog.	✓
.ldc	Lattice Design Constraint file	Used for specifying timing constraints for LSE synthesis flow. The .ldc contents will be combined into design database file .udb.	✓
.pcf	Power Calculator project file	Stores power analysis results from information extracted from the design project.	✓
.pdc	Post-Synthesis constraint file.	Used for specifying post-synthesis constraints (timing/physical) for Lattice engines such as MAP and PAR.	✓
.rdf	Radiant Software Project file	Used for managing and implementing all project files in the Radiant software.	✓
.rva	Reveal Analyzer file	Defines the Reveal Analyzer project and contains data about the display of signals in Waveform View.	✓
.rvl	Reveal Inserter debug file	Defines the Reveal project and its modules, identifies the trace and trigger signals, and stores the trace and trigger options.	✓
.sdc	SDC constraints file	Used for specifying design-specific constraints for Synplify-Pro. SDC is replaced by the FDC format in but is still supported in the Radiant software.	✓
.spf	Simulation project file, a script file produced by the Simulation Wizard	Used for running the simulator for your project from the Radiant software.	✓

Table 29: Source Files (Continued)

File Type	Definition	Function	Archive?
.sty	Strategy file	Defines the optimization control settings of implementation tools such as logic synthesis, mapping, and place and route.	✓
.sv	SystemVerilog		
.v	Verilog source file	Contains Verilog description of the circuit structure and function.	✓
.vhd	VHDL source file	Contains VHDL description of the circuit structure and function.	✓
.xcf	Configuration chain file	Used for programming devices in a JTAG daisy chain.	✓

Table 30: IP Files

File Type	Definition	Function	Archive?
<*instName*>_bb.v	Verilog IP black box file	Provides the Verilog synthesis black box for the IP core and defines the port list.	✓
<*instName*>.cfg	IP parameter configuration file	Used for re-creating or modifying the core in the IP Catalog tool.	✓
<*instName*>.svg	Scalable Vector Graphics file	A graphic file used to display module/IP schematic and ports.	✓
<*instName*>_tmpl.v	Verilog template file	A template for instantiating the generated module. This file can be copied into a user Verilog file.	✓
<*instName*>_tmpl.vhd	VHDL module template file	A template for instantiating the generated module. This file can be copied into a user VHDL file.	✓
<*instName*>.v	Verilog module file	Verilog netlist generated by IP Catalog to match the user configuration of the module.	✓

Table 31: Implementation Files

File Type	Definition	Function	Archive?
.bgn	Bitstream generation report file	Reports results of a bit generation (bitgen) run and displays information on options that were set.	
.bin	Bitstream file	Used for SRAM FPGA programming.	
.ibs	Post-Route I/O buffer information specification file (IBIS)	Used for analyzing signal integrity and electromagnetic compatibility (EMC) on printed circuit boards.	
.mcs	PROM file	Used for SRAM FPGA programming.	
.mrp	Map Report file	Provides statistics about component usage in the mapped design.	

Table 31: Implementation Files (Continued)

File Type	Definition	Function	Archive?
.pad	Post-Route PAD report file	Lists all programmable I/O cells used in the design and their associated primary pins.	
.par	Post-Route Place & Route report file	Summarizes information from all iterations and shows the steps taken to achieve a placement and routing solution.	
.sso	Post-PAR SSO analysis file	Reports the noise caused by simultaneously switching outputs.	
.tw1	Post-Map Timing analysis file	Estimates pre-route timing.	
.twr	Post-PAR Timing analysis file	Reports post-route timing.	
.vo	Post-Route Verilog simulation file	Used for post-route simulation.	
<*Design name*>_vo.sdf	Post-Route SDF simulation file for Verilog	Used for timing simulation.	
.vm	Synthesized netlist file	Netlist file generated by the Radiant software Synthesis tools.	
.udb	Unified Design Database file	Compiled from HDL design source. It may contain both design netlist and constraints.	

Revision History

The following table gives the revision history for this document.

Date	Version	Description
5/7/2020	2.1	▶ Updated to reflect changes in Radiant 2.1 software
11/27/2019	2.0	▶ Updated to reflect changes in Radiant 2.0 software. ▶ Removed Appendix A, Reveal User Guide. Starting with Radiant software v2.0, this is now a stand-alone user guide. ▶ Removed Appendix B, Programming Tools User Guide. Starting with Radiant software v2.0, this is now a stand-alone user guide.
03/25/2019	1.1	Updated to reflect changes in Radiant 1.1 software.
02/08/2018	1.0	Initial release.