Lattice Radiant Constraints Reference Guide

June 26, 2026

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

Introduction

Lattice Radiant™ is streamlined for smaller FPGAs but provides comprehensive constraint capabilities. Radiant offers advanced constraints operations such as timing constraints (multi-clock, domain crossing), physical pin assignments, constraint file management, and optimization goal settings.

This reference guide is designed to help FPGA designers understand, apply, and manage constraints within the Lattice Radiant™ design environment.

- **Timing Constraints**

  Specify complex timing constraints including multiple clocks, clock relationships, setup/hold times, and clock domain crossing constraints.

  Radiant provides static timing analysis and timing closure features to optimize and verify timing performance.

  Use the create_clock constraints for defining clock characteristics.

- **Physical Constraints**

  Create and edit pin assignments and physical port constraints.

  Radiant offers a device constraint editor that lets you assign pins and perform design rule checks.

  You can generate constraints for the synthesis and place-and-route implementation stages.

- **Constraint Files and Scripting**

  Radiant supports SDC constraint file formats, including the ability to define constraints in Tcl scripts.

  Radiant supports LDC constraint files and allows importing or creating physical constraints.

- **Optimization Goals**

  Radiant allows setting optimization goals like area or timing with constraint interpretation that guides synthesis optimization.

- **Constraint Editing and Validation**

  Radiant provides graphical and scripting interfaces for iterative constraint editing and checking.

  Radiant has features to validate constraint correctness through design rule checks.

Overview of Design Constraints in FPGA Design

Constraints are rules or directives applied to various elements of a digital design to ensure that the final implementation meets specific functional, performance, and integration goals. They serve as guidelines for synthesis, placement, routing, and other stages of the design flow.

Constraints are critical for:

- Achieving timing closure: Ensuring that all paths in the design meet required timing specifications.
- Managing reusable IP: Making sure that pre-designed blocks (IP cores) integrate correctly and perform as expected.
- Optimizing performance and area: Guiding tools for optimal trade-offs between speed, power, and silicon usage.

These are the general classifications of constraints in Radiant:

- Logical Constraints: These are typically specified in constraint files or through the GUI and affect synthesis and implementation.
- Physical Constraints: These relate to the physical implementation of the design on the FPGA device. They specify placement and routing instructions.
- HDL Attributes: These are embedded in the HDL source code and provide additional control over how specific elements are treated during synthesis and implementation.

Constraint Types

In this document, constraint type refers to the category or purpose of the constraint. It defines the aspect of the design that the constraint influences, such as timing behavior, physical pin assignments, or resource usage.

The constraint format, on the other hand defines how the constraints are written and conveyed to FPGA tools.

The main types of constraints are:

- **Timing Constraints**

  These constraints specify timing requirements. They define clock frequencies, setup and hold times, and path delays. They can be global or path-specific.

- **Physical Constraints**

  These constraints specify which physical pins or I/O banks on the FPGA are connected to which signals. They define electrical standards or characteristics of the pins, which is crucial for board-level integration and signal integrity.

- **Placement Constraints**

  These constraints direct where specific logic elements or blocks should be placed on the FPGA fabric.

- **Global Constraints**

  General device settings such as voltage, temperature, and family-specific options.

Constraint File Formats

Radiant supports the following constraint file formats, each with a specific purpose.

Types and Formats of Constraint Files

File Type	Description
.ldc	Lattice Design Constraints – general logical and timing constraints supported by Lattice Synthesis Engine (LSE).
.sdc	Synopsys Design Constraints – industry-standard timing constraints. These are supported by both LSE and Synplify Pro.
.pdc	Physical Design Constraints – pin location, placement, and routing.

Note

The current Radiant version still supports .fdc (FPGA Design Constraints) although this format may no longer be supported in future releases. It Synopsys SDC Standard timing constraints, such as create_clock, set_input_delay, and set_false_path, along with the non-timing constraints (design constraints), such as define_attribute, define_scope_collection, and define_io_standard.

Lattice Design Constraints (.ldc)

The .ldc (Lattice Design Constraints) is a unified constraint format used in Lattice Radiant to define timing, placement, and I/O constraints for FPGA designs. It combines the functionality of .sdc files into a single, more flexible format.

If you use an IP in the design, it uses .ldc files which will be consumed by both the tools, and these will be specific to the IP.

Purpose of .ldc Files

.ldc files are used to:

- Define clock constraints (period, waveform, generated clocks)
- Specify input/output delays
- Declare false paths, multicycle paths, and clock groups
- Assign pin locations, I/O standards, and drive strengths
- Control placement of logic elements (e.g., LOC constraints)

Key Features

.ldc Key Features

Feature	Description
Unified Format	Combines timing and physical constraints in one file
Tool Compatibility	The LDC contents will be written into the database which will then be supported throughout the flow, across synthesis, place-and-route, and timing analysis.
Human-Readable	Text-based, easy to edit and version control
Flexible Syntax	Supports SDC-like constructs

Examples

- Clock Definition
- Input Delay
- Output Delay
- False Path
- Pin Assignment

Best Practices for Using .ldc in Radiant

- Modularize: In the same .ldc file, group the constraints according to their function, such as clock constraints, IO constraints, and timing exception constraints.
- Validate: Radiant users use DRCs to check constraints and catch constraint errors.
- Document: Comment your constraints for clarity and maintainability
- Version Control: Track changes to constraints alongside HDL code.

Synopsys Design Constraints (.sdc)

The .sdc (Synopsys Design Constraints) file is an industry-standard format used to define timing constraints for digital designs. In Lattice Radiant, .sdc files are supported to provide timing intent to synthesis, place-and-route, and static timing analysis tools. This is supported by both LSE and Synplify Pro.

Purpose of .sdc Files

.sdc files are used to:

- Define clock characteristics
- Specify input and output delays
- Declare timing exceptions (such as false paths, multicycle paths)
- Group clocks for asynchronous domain handling

These constraints help the toolchain understand the timing environment of your design, enabling accurate optimization and validation.

Common .sdc Commands

.sdc Commands

Command	Description
create_clock	Defines a primary clock
set_input_delay / set_output_delay	Models external timing relationships
set_false_path	Ignores specific paths during timing analysis
set_multicycle_path	Used to define paths that take multiple clock cycles. It defines the number of cycles used by a clock, by a single clock or a clock across multiple clock domains.
set_clock_groups	Declares clocks as asynchronous

How .sdc Files Work in Radiant

- .sdc files are parsed during synthesis.
- They are not used for physical constraints like pin assignments or I/O standards (use .pdc).
- Radiant supports multiple .sdc files applied to the top level module.

  Current support is only for synthesis.

Best Practices for Using .sdc in Radiant

- Keep timing constraints separate from physical constraints (e.g., use timing.sdc alongside io.pdc)
- Use descriptive names for clocks and ports to improve readability
- Validate constraints using Radiant software's timing reports and constraint checker
- Avoid redundancy with .ldc files if both are used—prefer .ldc for unified constraint management. (Only supported for LSE flow.)

Note

The .sdc format is supported for both LSE and Synplify Pro. The .ldc format is supported only for Synplify Pro.

When to Use .sdc vs .ldc

Using .sdc and .ldc

Use Case	Recommended Format
Timing-only constraints	.sdc or .ldc
Unified timing + physical constraints	.ldc
Legacy or cross-tool compatibility	▶ .sdc is for both LSE and Synplify Pro. ▶ .ldc is only for Synplify Pro

Physical Design Constraints (.pdc)

The .pdc (Physical Design Constraints) file is a constraint format used in Lattice Radiant to define low-level physical implementation constraints. These constraints are primarily used to control placement, routing, and floorplanning of logic elements within the FPGA fabric.

Purpose of .pdc Files

.pdc files allow designers to:

- Precisely locate logic elements (e.g., flip-flops, LUTs, RAMs) in specific regions or tiles
- Define placement regions for modules or logic groups
- Control routing paths or restrict routing resources
- Apply timing-driven placement constraints

These constraints are especially useful for:

- Floorplanning large or hierarchical designs
- Optimizing timing by controlling logic proximity
- Meeting power or area goals by grouping logic

Integration in Radiant

- .pdc files are used after synthesis, during map and place and route.
- They are used for timing and or I/O constraints.
- You can add .pdc files to your project via Project > Add Files or include them in a TCL script.

Best Practices for Using .pdc Files

- Combine with logical hierarchy: Use module names or instance paths to apply constraints cleanly.
- Validate with floorplan viewer: Use Radiant software's GUI tools to visualize and verify placement.
- Too many physical constraints can limit optimization and increase routing congestion.
- Document intent: Use comments to explain why a component is placed in a specific region.

When to Use .pdc and .ldc

.pdc and .ldc Usage

Use Case	Recommended Format
Pin assignments, I/O standards	ldc/pdc
Logic placement and floor planning	pdc
Routing restrictions	pdc

Constraints Methodology

The flow of constraint entry in Lattice Radiant follows a structured process that aligns with the stages of the FPGA design flow. This ensures that constraints are applied at the right time and in the right format to guide synthesis, placement, routing, and timing analysis effectively.

Here is a breakdown of the typical flow:

1. Design Initialization
   a. Create or import your project in Radiant.
   b. Add your HDL source files, IP cores, and any existing constraint files.
   c. Define the target device and top-level module.
2. Pre-Synthesis Constraints Entry

   This is where you define constraints that influence synthesis and early design analysis.

   Common Pre-Synthesis Constraints:

   - Timing constraints (such as primary clocks, input/output delays)
   - Timing exception constraints (such as set_false_path, set_multicycle_path)
   - HDL attributes (embedded in Verilog/VHDL)

   Tools Used:

   - Pre-synthesis Constraints Editor
   - Text-based constraint files (such as .sdc, .ldc)
3. Post-Synthesis Constraint Entry

   After synthesis, additional constraints can be applied to guide placement and routing.

   Common Post-Synthesis Constraints:

   - Physical constraints (such as pin assignments, location constraints)
   - Routing directives (such as area groups, prohibited regions)

   Tools Used:

   - Device Constraint Editor
   - Post-synthesis Timing Constraints Editor
   - Physical Designer
4. Design Flow Integration

   Constraints are integrated at various stages:

   - Synthesis: Uses logical and timing constraints to optimize logic.
   - Map, Place, and Route: Uses timing and physical constraints to guide layout.
   - Timing Analysis: Validates that all timing constraints are met.

5. Validation and Debug

   After implementation, Radiant provides tools to:

   - Check constraint validity
   - Generate timing and placement reports
   - Debug violations or conflicts

Pre-synthesis versus Post-synthesis Constraints

In Lattice Radiant, the pre-synthesis and post-synthesis flows refer to different stages in the FPGA design process.

Pre-Synthesis Flow

This stage involves everything before the design is synthesized into a gate-level netlist.

Key Activities:

- Design Entry: Writing HDL code (VHDL/Verilog/SystemVerilog).
- Constraint Definition: Creating SDC files for timing and placement constraints.
- Simulation: Running functional simulations to verify logic correctness.
- Elaboration: Checking for syntax and semantic correctness.
- Linting/Static Analysis: Optional checks for code quality and potential issues.

Purpose:

- Ensure the design is logically correct and meets initial constraints.
- Catch errors early before synthesis.

Post-Synthesis Flow

This stage starts after the synthesis tool has converted the HDL into a gate-level netlist.

Key Activities:

- Synthesis Output Review: Analyze the netlist, resource usage, and timing estimates.
- Implementation: Placement and routing of the design onto the FPGA fabric.

- Timing Analysis: Static timing analysis (STA) to ensure timing constraints are met.
- Bitstream Generation: Creating the final file to program the FPGA.
- Post-Implementation Simulation: Optional simulation using the netlist and timing data.

Purpose:

- Ensure the design meets timing, area, and power requirements.
- Prepare the design for actual hardware deployment.

Design Flow Integration Points

Understanding how and where constraints integrate into the design flow helps ensure your FPGA design meets functional and performance goals.

Here's a detailed breakdown of design flow integration points for constraints in Radiant:

Design Entry Stage

- Constraint Type: Logical constraints
- Files: HDL source files or .sdc/.ldc (Synopsys Design Constraints)
- Purpose: Define clocks, I/O standards, pin locations, and basic timing requirements.
- Integration Point: Constraints are referenced during elaboration and synthesis to guide logic optimization and resource mapping.

Synthesis Stage

- Constraint Type: Timing and resource constraints
- Files: .sdc/.ldc used by the synthesis engine
- Purpose: Ensure the synthesized netlist respects timing budgets.
- Integration Point: Constraints influence how logic is mapped to FPGA primitives and how clocks are propagated.

Post-Synthesis Analysis

- Constraint Type: Timing constraints
- Tools: Timing Analyzer
- Purpose: Validate that the synthesized design meets setup/hold requirements.
- Integration Point: Constraints are used to generate timing reports and identify violations.

Place and Route (P&R)

- Constraint Type: Physical and timing constraints
- Files: .pdc
- Purpose: Guide the placement of logic blocks and routing of signals to meet timing and I/O requirements.
- Integration Point: Use sparingly: Only apply physical constraints when necessary to meet timing or layout goals.

Post-Implementation Timing Analysis

- Constraint Type: Final timing constraints
- Tools: Static Timing Analyzer
- Purpose: Verify that the implemented design meets all timing constraints under worst-case conditions.
- Integration Point: Final constraints are used to generate timing reports.

Bitstream Generation

- Constraint Type: Finalized constraints
- Purpose: Ensure the programmed device behaves as expected with correct I/O configuration and timing.
- Integration Point: Constraints are embedded in the bitstream metadata.

Summary of Integration Points

Summary of Integration Points

Stage	Constraint Role
Design Entry	Define clocks, I/O, basic timing
Synthesis	Guide logic mapping and timing
Post-Synthesis Analysis	Validate timing of synthesized netlist
Place and Route	Optimize physical layout for timing
Post-Implementation STA	Final timing verification
Bitstream Generation	Embed final constraints

Organizing Constraints

In complex FPGA designs, it's common to split constraints across multiple files for better organization, reuse, and collaboration. Lattice Radiant supports this modular approach, but it’s important to manage these files correctly to avoid conflicts and ensure proper constraint application.

Note

In Radiant version 2025.2, this is only supported for top level module files.

Why use multiple constraint files?

- Separation of Concerns: Keep clock definitions, I/O assignments, and timing constraints in separate files.
- Team Collaboration: Different team members can work on different constraint domains.
- IP Reuse: Reuse constraints associated with IP blocks or modules.
- Design Scalability: Easier to manage and debug in large designs.

How to Manage Multiple Constraint Files

1. Set File Priority and Order
   - Radiant applies constraints in the order they appear in the project.
   - Later files override earlier ones if there are conflicting constraints.
   - Use this to your advantage by placing general constraints first and overrides later.
2. Use Comments and Naming Conventions
   - Name files clearly (e.g., clocks.sdc, io_assignments.sdc, timing_exceptions.sdc).
   - Add comments to explain the purpose of each constraint block.
3. Avoid Conflicts
   - Do not define the same clock or pin in multiple files unless intentionally overriding.
4. Validate Constraints
   - Use the Constraint Checker in Radiant to detect:
     - Duplicate or conflicting constraints
     - Unused or undefined nets
     - Syntax errors

Best Practices

- Version Control: Track constraint files in Git or another VCS for traceability.
- Run timing analysis early. Run timing analysis post synthesis to catch issues before implementation.
- Document Assumptions: Especially for timing exceptions such as false paths or multicycle paths.
- Modularize by Function: Group constraints by subsystem or IP block.

Project Directory Structure and File Naming Conventions

Maintaining a clean and consistent project structure is essential for scalability, collaboration, and debugging in FPGA design. This is especially true when managing multiple constraint files in Lattice Radiant.

Recommended Project Directory Structure

```
/my_fpga_project/
│
├── /src/                   # HDL source files (VHDL,
Verilog, SystemVerilog)
│   ├── top.v
│   └── modules/
│       └── uart.v
│
├── /constraints/           # All constraint files
│   ├── clocks.ldc
│   ├── io.ldc
│   ├── timing.ldc
│   ├── constraints.sdc     # Optional: SDC file for
compatibility
│
├── /sim/                   # Simulation files and
testbenches
│   ├── tb_top.v
│   └── waveforms/
│
├── /impl/                  # Implementation outputs
│   ├── netlist/
│   ├── reports/
│   └── bitstream/
│
├── /scripts/               # TCL scripts or automation tools
│
└── my_fpga_project.rdf     # Radiant project file
```

Best Practices

- Group by Function: Keep constraints modular (for example, separate clocks from I/O).
- Use Lowercase with Underscores: Improves readability and consistency across platforms.
- Avoid Spaces: Use underscores (_) instead of spaces in file names.
- Version Control: Track all constraint files in Git or another VCS.
- Document Inside Files: Use comments (# or //) to explain each constraint block.

Integration Tip

In Radiant, you can add multiple constraint files via:

- GUI: Project > Add Files
- TCL Script: Use add_file or add_constraints commands
- Project File (.rdf): Ensure all constraint files are listed and ordered correctly

Adding Timing Constraints

Defining Primary Clocks and Virtual Clocks

In FPGA design, clocks are the backbone of timing analysis. Lattice Radiant supports defining both primary clocks and virtual clocks using constraint files such as .ldc or .sdc. Proper clock definition is essential for accurate static timing analysis (STA) and timing closure.

Clock Types

Clock Type	Description
Primary Clock	A real, physical clock signal that drives sequential elements in the design.
Virtual Clock	A conceptual clock that is defined without being attached to any physical netlist element, pin, or port in the design.

Defining Clocks in .ldc or .sdc

Primary Clock Example

```
create_clock -name clk -period 10.0 -waveform {0 5} [get_ports clk]
```

This tells the tool that clk is a real clock with a 10 ns period (100 MHz).

Virtual Clock Example

```
create_clock -name ext_clk -period 20.0 -waveform {0 10}
```

This clock is not tied to a physical port but is used to model timing for external signals.

When to Use Virtual Clocks

- For input delay modeling when the data comes from an external device not driven by your internal clock.
- For output delay modeling when your FPGA drives data to an external device with its own clock.
- For asynchronous interfaces where no internal clock is available to reference.

Input/Output Delay with Virtual Clocks

These constraints help the tool understand the timing relationship between your FPGA and external systems.

Best Practices

- Always name your clocks clearly (sys_clk, virt_clk, clk_100m) for readability.
- Use virtual clocks for clean separation of internal and external timing domains.
- Validate clock definitions using Radiant's timing reports to ensure they are applied correctly.
- Avoid duplicate clock definitions across multiple constraint files.

Input and Output Delay Constraints

In digital design, input and output delay constraints are critical for ensuring that the timing of signals entering and leaving the FPGA meets system-level requirements. Lattice Radiant provides a robust environment for specifying and analyzing these constraints to achieve timing closure.

- Input Delay: Specifies the time between when a signal becomes valid at the FPGA input pin and when it is expected to be captured by the internal logic. This accounts for delays introduced by external devices or board-level routing.
- Output Delay: Defines the time between when a signal is launched from the FPGA and when it is expected to be captured by an external device. This includes internal routing and clock-to-output delays.

Specifying Delay Constraints

In Radiant, delay constraints are typically defined using the constraints editor or directly in the .sdc file. The relevant commands are:

- -clock: Specifies the clock domain for the delay.
- The delay value (e.g., 5.0) is in nanoseconds.
- get_ports: Identifies the input or output port.

Best Practices

- Match system timing: Align delay values with the timing characteristics of external components.
- Use realistic margins: Account for board-level variations and temperature/voltage fluctuations.
- Validate with timing analysis: Use Radiant's Timing Analysis View to verify that all paths meet setup and hold requirements.

Visualizing Constraints

Radiant provides graphical tools to visualize timing paths and constraint coverage. Use the Timing Reports and Path Details views to inspect how input and output delays affect overall timing

Generated Clocks

In FPGA designs, generated clocks are clocks derived from a base or primary clock using internal logic such as PLLs (Phase-Locked Loops) or clock dividers. Accurately defining these clocks in Radiant is essential for correct timing analysis and constraint propagation.

What Is a Generated Clock?

A generated clock is a clock signal that is not directly driven by an external source but is instead created within the FPGA fabric. Examples include:

- PLL output clocks
- Clock dividers or multipliers
- Clocks gated or muxed by logic

Defining Generated Clocks

Generated clocks are specified using the create_generated_clock constraint in the .sdc file. This informs the timing engine how the new clock relates to its source.

Example: PLL Output Clock

```
create_generated_clock -name pll_clk_out -source clk_in -
divide_by 2 [get_pins {pll_inst/CLKOUT}]
```

- -name: Assigns a name to the generated clock.
- -source: Specifies the source clock or port.
- -divide_by: Indicates the frequency division factor.
- [get_pins]: Identifies the output pin of the clock-generating element.

Example: Clock from Logic

```
create_generated_clock -name logic_clk -source clk_in -
divide_by 1.5 [get_pins {logic_inst/clk_out}]
```

This defines a clock that is 2/3 the frequency of the source clock, generated through logic.

Radiant timing engine automatically creates “generated clocks”. You must define your input clock to the PLL and clock dividers.

Best Practices

- Allow Radiant timing engine to generate the clocks automatically for PLLs or clock to error.
- If generic clock constraints are defined manually, use meaningful names for clarity in timing reports.
- Verify clock relationships using the Clock Report in Radiant’s Timing Analysis View.

Reporting Generated Clocks

Radiant provides reports to inspect and validate clock definitions:

- Clock Summary report: Lists all clocks, including generated ones, with their frequencies and sources.
- The relationships and domain crossing between clocks are also shown as part of the clock report.

Clock Groups and Bus Skew

In complex FPGA designs, managing multiple clock domains and ensuring timing consistency across wide data buses is essential. Lattice Radiant provides constraint mechanisms to define clock groups and control bus skew, enabling accurate timing analysis and robust design implementation.

Clock Groups

Clock groups are used to inform the timing engine about the relationship or lack thereof between different clocks in the design. This is especially important when clocks are asynchronous or unrelated, as it prevents false timing violations across domains.

Types of Clock Group Relationships

- Asynchronous: No timing relationship; paths between these clocks are ignored.
- Physically Exclusive: Clocks never toggle at the same time (e.g., muxed clocks).
- Logically Exclusive: Clocks are mutually exclusive by design logic.

Defining Clock Groups

Use the set_clock_groups constraint in the SDC file:

```
set_clock_groups -asynchronous -group {clk_a} -group {clk_b}
```

This tells the timing analyzer to ignore paths from clk_a to clk_b and vice versa.

Bus Skew

Bus skew refers to the timing variation between signals in a parallel data bus. Excessive skew can lead to setup or hold violations, especially at high frequencies or across long routing paths.

Managing Bus Skew

Radiant allows designers to constrain and analyze skew across bus signals using:

- Max Skew Constraints: Limit the allowable skew between signals.

Example: Skew Constraint

```
set_max_skew [get_pins A] 0.25
```

This constraint requires the maximum difference between the net delays of the loads on the net connected to pin “A” to be less than or equal to 250 picoseconds.

Best Practices

- Always define clock groups for unrelated clocks to avoid false path analysis.
- Use skew constraints for wide buses, especially in high-speed interfaces like DDR, LVDS, or SERDES.
- Review timing reports to identify and resolve skew-related violations.

Tools for Analysis

- Clock interaction report: Reports relationships between clock domains.
- Skew report: Highlights skew across defined bus groups.
- Path details view: Allows inspection of individual timing paths for skew and cross-domain issues.

Multicycle and False Path Constraints

In FPGA designs, not all timing paths require analysis on every clock cycle. Some paths are intentionally designed to take multiple cycles, while others are never expected to transfer data. Radiant supports multicycle and false path constraints to help designers accurately model these scenarios and avoid misleading timing violations.

A multicycle path is a timing path that is allowed to take more than one clock cycle to propagate data. This is common in pipelined or staged logic where data is intentionally delayed.

Multicycle Path

Use the set_multicycle_path constraint in the .sdc file:

- This line sets the setup requirement to 2 cycles.

```
# Setup requirement: 2 cycles
set_multicycle_path 2 -setup -from [get_clocks clk_a] -to
[get_clocks clk_b]
```

- This line adjusts the hold requirement accordingly (typically N-1).

```
# Hold requirement: 1 cycle (typically N-1)
set_multicycle_path 1 -hold -from [get_clocks clk_a] -to
[get_clocks clk_b]
```

Use Cases

- Clock domain crossing paths
- Deeply pipelined data paths
- State machines with slow transitions
- Low-priority control signals

False Path

A false path is a timing path that will never be exercised during normal operation. These paths are excluded from timing analysis to prevent false violations.

You can also apply false paths between specific registers or ports:

Use Cases

- Asynchronous domain crossings
- Debug or test logic
- Muxed paths that are never active simultaneously

Best Practices

- Use false paths only when you are certain the path is non-functional.
- Document the rationale for each constraint to aid future maintenance and reviews.

Tools for Verification

- Timing Reports: Confirm that false paths are correctly excluded or adjusted.
- Path Details View: Inspect individual paths to verify constraint application.

Max/Min Delay Constraints

Precise control over timing paths is essential in FPGA design, especially when dealing with non-standard timing requirements or optimizing for performance. Radiant allows designers to apply max/min delay constraints and setup/hold adjustments to fine-tune timing analysis and ensure robust operation.

Max delay and min delay constraints define the allowable timing window for signal propagation between two points. These constraints are useful for controlling data arrival times, enforcing timing margins, or modeling external device behavior.

Syntax Examples

- set_max_delay: Limits the longest allowable path delay.
- set_min_delay: Ensures a minimum delay, useful for avoiding hold violations.

Use Cases

- Interface timing with external devices
- Timing control in asynchronous logic
- Enforcing design-specific timing margins

Setup/Hold Adjustments

Setup and hold adjustments modify the default timing requirements for data arrival relative to clock edges. These are typically used to model clock uncertainty, board-level skew, or intentional timing offsets. Here is a focused example illustrating setup and hold timing constraints:

```
tcl
# Define the clock with 20ns period (50MHz)
create_clock -period 20 -name clk [get_ports clk]

# Setup constraint: maximum input delay on data_in relative to
clk
set_input_delay -clock clk -max 5.0 [get_ports data_in]

# Hold constraint: minimum input delay on data_in relative to
clk
set_input_delay -clock clk -min 1.0 [get_ports data_in]

# Output delays similarly can be set
set_output_delay -clock clk -max 5.0 [get_ports data_out]
set_output_delay -clock clk -min 1.0 [get_ports data_out]

# Multicycle path example for setup (start edge) and hold
set_multicycle_path 3 -setup -start -from [get_clocks clk] -to
[get_clocks clk]
set_multicycle_path 2 -hold -from [get_clocks clk] -to
[get_clocks clk]
```

Explanation:

create_clock defines your reference clock.

set_input_delay -max sets the setup timing (latest arrival time).

set_input_delay -min sets the hold timing (earliest arrival time).

set_multicycle_path adjusts the number of clock cycles for setup and hold checks (useful for multicycle paths).

Use Cases

- Accounting for jitter and skew
- Tightening or relaxing timing margins
- Modeling worst-case scenarios

Best Practices

- Use max/min delay constraints only when necessary to enforce specific timing behavior.
- Adjust setup/hold margins based on board-level analysis and simulation data.
- Validate constraints using Radiant's Timing Analyzer and path reports.

Tools for Analysis

- Timing reports: Show paths affected by max/min delay and setup/hold adjustments.
- Constraint coverage report: Ensures all critical paths are properly constrained.
- Path details view: Allows inspection of individual timing paths and applied margins.

Macro Placement through Physical Constraints

In Radiant, macro placement is handled through physical constraints that guide the location of logic elements within the FPGA fabric. This chapter shows how macro placement of constraints in Radiant is achieved using physical design constraints (.pdc). This involves a multi-step process aimed at defining regions for macro blocks in your FPGA design, setting placement constraints, and finally applying these constraints during place and route stages.

1. Define a macro region in Physical Designer where the macro will be physically placed on the FPGA.
2. Save the macro regions and related constraints into a post-synthesis constraint file (.pdc).
3. The placement constraints are applied in Place & Route. When place and route (PAR) process is run, the tool obeys the placement constraints defined in the .pdc file, placing the macro in the specified region.
4. After place and route, export your macro with different preservation levels (Logical, Firm with placement, Hard with placement and routing). Exported macros can be reused in other projects with their placement constraints.

Examples

- Macro creation constraint (in .sdc or .pdc):

  ```
  ldc_create_macro -name macro_name
  ```

  This command defines the macro by name during the pre-synthesis phase.

- Macro region constraint (post-synthesis, in .pdc file):

  ```
  ldc_create_region -name macro_region_name -device
  device_name -bbox {x1 y1 x2 y2} -exclusive
  ```

  -bbox {x1 y1 x2 y2} specifies the rectangular floorplan coordinates to constrain the macro placement.

  -exclusive optionally excludes other logic from occupying this region.

- Linking Macro to Region:

  ```
  ldc_assign_macro_region -macro macro_name -region
  macro_region_name
  ```

  This associates the defined macro with the placed region on the device.

- Here is an example of a macro named "myMacro" placed in a region named "myRegion":

  ```
  ldc_create_macro -name myMacro

  ldc_create_region -name myRegion -device LFE5UM5G-85F -bbox
  {50 100 150 200} -exclusive
  ```

```
ldc_assign_macro_region -macro myMacro -region myRegion
```

These constraint lines are typically generated or edited in the Physical Designer tool after synthesis and saved in the .pdc file for placement during the place and route steps.

The bounding box coordinates correspond to the FPGA site grid locations and should be adjusted based on resource usage and floorplanning needs.

This example flow and syntax are consistent with the Radiant macro placement methodology where pre-synthesis macro definition (ldc_create_macro) combines with post-synthesis physical region definition (ldc_create_region), and the assignment of macro to region for placement control.

For more information in uing macro blocks, see the Radiant software help topic: User Guides > Entering the Design > Block-Based Design - Using Macro Blocks

Constraints Usage in Synthesis and Implementation

How Synthesis Engines Use Constraint Files

In Radiant, constraint files play a critical role in guiding the synthesis engine during the transformation of RTL code into a gate-level netlist. These files provide essential information that influences how the design is interpreted, optimized, and mapped to the target FPGA device.

Purpose of Constraint Files in Synthesis

Constraint files contain directives that inform the synthesis engine about:

- Timing Requirements: Specifies clock definitions, input/output delays, and multi-cycle paths to ensure the design meets performance goals.
- Physical Constraints: Includes pin assignments, I/O standards, and placement directives that affect how logic is mapped to the physical FPGA resources.
- Design Intent: Helps the synthesis engine understand critical paths, false paths, and areas where optimization should be focused or relaxed.

How Constraints Influence Synthesis

- Clock and Timing Analysis

  The synthesis engine uses clock definitions and timing constraints to build a timing model. This model guides logic optimization, such as retiming and pipelining, to meet setup and hold requirements.

- Resource Mapping

  Physical constraints like pin locations and region assignments influence how logic blocks are placed and routed. This ensures compatibility with board-level design and helps reduce routing congestion.

- Optimization Directives

  Constraints can specify areas of the design that require high performance or low power, allowing the synthesis engine to apply targeted optimizations.

- Validation and Feedback

  During synthesis, the engine checks constraints for consistency and feasibility. Invalid or conflicting constraints are flagged, allowing designers to correct issues early in the flow.

Workflow Integration

Constraint files are typically loaded or referenced during project setup and are used throughout the synthesis and implementation flow. They can be edited using tools like:

- Device Constraint Editor
- Physical Designer
- Text Editor (for manual editing)

Interaction Between Constraint Files and Optimization Settings

In Radiant, constraint files and optimization settings work together to guide the synthesis and implementation tools toward producing a design that meets performance, area, and power goals. Understanding how these elements interact is key to achieving efficient and reliable FPGA designs.

Role of Constraint Files

Constraint files define the design intent by specifying:

- Timing constraints (e.g., clock definitions, input/output delays)
- Physical constraints (e.g., pin assignments, region placement)
- Design rules (e.g., false paths, multi-cycle paths)

These constraints inform the tools about what aspects of the design are critical and where flexibility exists.

Role of Optimization Settings

Optimization settings in Radiant control how aggressively the tools attempt to improve various aspects of the design, such as:

- Timing performance
- Area utilization
- Power consumption
- Routing congestion

These settings can be adjusted globally or locally, and they influence how the synthesis and place-and-route engines interpret and act on the constraints.

Key Interactions

- Timing-Driven Optimization

  When timing constraints are defined, the synthesis engine prioritizes meeting setup and hold requirements. Optimization settings like “High Performance” or “Balanced” determine how much effort is spent on retiming, logic duplication, and pipelining.

- Placement and Routing Guidance

  Physical constraint files guide the placement engine. Optimization settings can influence how tightly logic is packed or spread out to meet timing or reduce power.

- Conflict Resolution

  If constraints conflict with optimization goals (e.g., tight placement constraints vs. timing requirements), Radiant uses heuristics to balance them. Warnings or errors may be issued to prompt user intervention.

- Incremental Optimization

  Constraints can be used to lock down parts of the design, allowing optimization to focus on other areas. This is useful in iterative design flows where stability in certain blocks is required.

Best Practices

- Align Constraints with Optimization Goals: Ensure that timing and physical constraints reflect the actual performance and layout requirements.
- Use Constraint Validation Tools: Radiant provides feedback on constraint feasibility—use this to refine your constraints before synthesis.
- Iterate and Refine: Optimization is not one-size-fits-all. Adjust settings and constraints based on timing reports and resource usage feedback.

Mapping, Place and Route Considerations

The mapping, placement and routing stages transforms synthesized logic into a fully implemented design ready for bitstream generation. Each stage is influenced by constraints, optimization settings, and device architecture.

Mapping

Mapping refers to the process of translating the synthesized netlist into FPGA-specific primitives such as LUTs, flip-flops, and I/O buffers.

Key Considerations:

- Technology Mapping: Ensures that generic logic is converted into device-compatible elements.
- Constraint Influence: Timing and physical constraints guide how logic is grouped and prepared for placement.
- Resource Awareness: Mapping considers available resources (e.g., DSP blocks, RAMs) and attempts to use them efficiently.

Placement

Placement assigns mapped logic elements to specific physical locations on the FPGA.

Key Considerations:

- Timing Optimization: Critical paths are placed to minimize delay and meet timing constraints.
- Constraint Compliance: Placement respects region constraints, pin assignments, and locked blocks defined in Physical Designer.
- Congestion Management: The placer avoids dense areas to reduce routing complexity and improve timing.

Routing

Routing connects the placed elements using the FPGA’s interconnect fabric.

Key Considerations:

- Timing Closure: Routing algorithms prioritize paths that are timing-critical, using shortest or fastest routes.
- Signal Integrity: Ensures that routing meets electrical constraints such as drive strength and slew rate.
- Constraint Validation: Confirms that routed paths comply with all physical and timing constraints.

Best Practices

- Use Floorplanning Tools: Tools like Physical Designer help guide placement for better timing and resource usage.
- Review Reports: After place and route, examine timing, utilization, and routing reports to identify bottlenecks.
- Iterate with Constraints: Adjust constraints based on post-route analysis to improve performance or resolve violations.

Evaluating Results

The Mapping, Placement, and Routing stages in Radiant convert synthesized logic into a physical design that meets timing, area, and power constraints. Each stage is influenced by user-defined constraints and optimization settings, and the results are evaluated using detailed reports.

Sample Timing Reports

Timing reports are generated after Place and Route and provide insight into whether the design meets its timing goals. Below are examples of key sections from a typical timing report:

- Clock Summary

The clock summary report shows defined clocks, their periods, and where they originate. Helps verify that clock constraints were correctly applied.

Table 1: Clock Summary

Clock Name	Period	Frequency	Source	Used Resources
clk_main	5.000 ns	200.0 MHz	Pin CLK_IN	120 FFs, 45 LUTs
clk_aux	10.000 ns	100.0 MHz	PLL_OUT	60 FFs, 30 LUTs

- Worst Path Timing Analysis

  The data indicates that the path meets timing (positive slack). If slack were negative, the design would fail timing and require optimization.

  Path Group: clk_main

  Startpoint: U1/data_reg[3]

  Endpoint: U2/output_reg[3]

  Path Delay: 4.850ns

  Required Time: 5.000ns

  Slack: +0.150ns

- Setup and Hold Violations

  Hold violations may require changes in placement, routing, or clock skew adjustments.

Setup Violations: 0

Hold Violations: 2

Hold Violation Example:

Startpoint: U3/control_reg[1]

Endpoint: U4/status_reg[1]

Hold Slack: -0.120 ns

- Timing Summary

  The data provides a high-level view of timing performance across the design.

  Total Paths Analyzed: 1,245

  Paths Failing Setup: 0

  Paths Failing Hold: 2

  Max Delay: 4.95 ns

  Min Delay: 0.85 ns

Best Practices

- Review Slack Values: Positive slack means timing is met; negative slack requires attention.
- Use Physical Constraints: Guide placement to reduce long paths and improve timing.
- Iterate with Optimization Settings: Adjust synthesis and place-and-route settings to resolve violations.

Timing Exceptions and Advanced Scenarios

Exceptions: False Paths, Asynchronous Paths

In FPGA design, not all signal paths are relevant for timing analysis. To ensure accurate timing closure and efficient optimization, designers can use exceptions to exclude certain paths from timing checks. These exceptions help the tools focus on meaningful timing paths and avoid misleading violations.

False Paths

False paths are signal paths that exist in the design but are never activated during normal operation due to design logic or control conditions.

Use Case:

- Paths between mutually exclusive states in a finite state machine (FSM).
- Debug or test logic that is not active during runtime.

How to Declare:

In Radiant, declaring false paths in .sdc (Synopsys Design Constraints) files is done using the standard set_false_path command, which is part of the SDC format supported by Radiant. Here's how you can use it:

```
set_false_path -from [get_ports {source_port}] -to [get_ports {destination_port}]
```

You can also use internal nets or clocks:

```
set_false_path -from [get_clocks clk1] -to [get_clocks clk2]
```

Notes

- Constraint Coverage: By default, set_false_path does not improve constraint coverage in Radiant. However, starting with Radiant 2023.1, you can use the -false_covers timing command line option to include false paths in timing coverage reports 1.
- Strategy Settings: You can apply the -false_covers option in the Strategy settings of your project to ensure these paths are considered during timing analysis.
- Internal Nets: If you are targeting internal nets rather than ports or clocks, use appropriate selection methods like get_cells, get_nets, or get_pins to specify the path endpoints. For example: set_false_path -from [get_pins {U1/clk}] -to [get_pins {U2/data}].

Impact:

- Excluded from setup and hold analysis.
- Helps prevent unnecessary timing violations and over-optimization.

Asynchronous Paths

Asynchronous paths connect logic driven by different, unrelated clocks. These paths are not timed in the traditional sense and require special handling to avoid false violations.

Use Case:

- Data transfer between clock domains.
- Handshake or FIFO interfaces.

How to Declare:

To declare asynchronous paths in .sdc files, the recommended and most robust method is to use the set_clock_groups command with the -asynchronous option. This informs the timing analyzer that paths between these clock domains should not be timed, as they are asynchronous and not expected to have a deterministic timing relationship.

Recommended Syntax

```
set_clock_groups -asynchronous \
  -group {CLK_A CLK_B} \
  -group {CLK_X} \
  -group {CLK_Y}
```

This example declares these asynchronous clock groups:

- CLK_A and CLK_B are synchronous with each other.
- CLK_X and CLK_Y are each in their own asynchronous group.

Why use set_clock_groups -asynchronous?

- It automatically disables timing analysis between the specified groups in both directions.
- It avoids the need to manually declare set_false_path for each direction between clocks.
- It provides clear documentation of clock relationships for the tool and future maintainers.

Best Practices

- Use exceptions as long as they do not hide real timing issues.
- Document assumptions: Clearly annotate why a path is marked false or asynchronous.
- Verify with simulation: Ensure that excluded paths are truly inactive or asynchronous in functional behavior.

Upgrading and Migrating Constraint Files

When transitioning between tool versions or migrating designs from other environments, it's important to ensure that constraint files are compatible and correctly interpreted by Radiant. This section outlines best practices and common steps for upgrading and migrating constraint files.

Migrating from Other Vendor Tools

To migrate constraints files from other vendors:

1. Convert object names to match Radiant's naming conventions.
2. Remove unsupported commands (e.g., vendor-specific attributes).
3. Use create_clock and set_false_path in .sdc format for timing.
4. Test constraints incrementally to isolate compatibility issues.

Upgrading Between Radiant Versions

Considerations:

- New versions may introduce updated constraint syntax or validation rules.
- Deprecated commands may trigger warnings or errors.

Best Practices:

- Review release notes for constraint-related changes.
- Re-validate all constraints after upgrading.
- Use GUI editors to regenerate or verify constraints visually.

Tips for Smooth Migration

- Back up original constraint files before editing.
- Use comments to document changes and assumptions.
- Modularize constraints (e.g., separate timing and physical files).
- Test incrementally: Apply constraints in stages and validate after each.

Migrating from Lattice Diamond Preferences

Lattice Design Constraints (.ldc/.pdc) are a combination of both Synopsys Design Constraints and the Radiant software proprietary physical constraints.

All Lattice Diamond constraints were known as Preferences and were defined as .ldc or .pdc files written as .sdc constraints. Lattice Diamond uses a combination of “preferences” and constraints, while Radiant software uses only .ldc constraints. Lattice Diamond preferences that can be supported as constraints in the Radiant software are listed in the Help section:

User Guides > Applying Design Constraints > Migrating from Former Lattice Diamond Preferences

Validation and Debug

Running Constraints Checks

In FPGA designs, especially those involving PLLs, clock dividers, or clock multiplexers, generated clocks and multiple clock domains are common. Radiant supports constraint definitions that help the synthesis and timing engines correctly analyze these complex clocking scenarios.

Generated Clocks

Generated clocks are clocks derived from a primary clock source using logic elements like PLLs, clock dividers, or gating circuits.

Why They Matter:

- Timing analysis must understand the relationship between the source and generated clocks.
- Without proper constraints, tools may misinterpret timing paths or fail to analyze them correctly.

How to Define:

Use the create_generated_clock command in .sdc files.

Best Practices:

- Always specify the source clock.
- Use meaningful names for clarity in reports.
- Validate generated clock definitions using timing reports.

Multiple Clock Domains

Designs with multiple clock domains require careful constraint management to ensure accurate timing analysis and safe data transfer between domains.

Key Concepts:

- Clock Domain Crossing (CDC): Paths between different clock domains must be treated as asynchronous.
- Domain Isolation: Timing paths should be constrained or excluded to avoid false violations.

How to Handle:

Use .sdc constraints to isolate domains.

Best Practices:

- Identify all clock domains early in the design.
- Use synchronization techniques (e.g., FIFOs, handshake logic) for CDC paths.
- Review timing reports to confirm that asynchronous paths are excluded.

Validation and Debugging

After applying constraints:

- Use Timing Analyzer to verify clock relationships.
- Check for missing or misconfigured clocks in reports.
- Ensure no unintended paths are analyzed between asynchronous domains.

Reviewing and Interpreting Reports

After applying constraints and running synthesis, placement, and routing in Lattice Radiant, it's essential to review the generated reports to ensure the design meets timing, resource, and functional requirements. These reports provide critical feedback on how well the design adheres to the specified constraints.

Timing Reports

Timing reports help verify whether the design meets setup and hold requirements for all defined clocks.

Key Sections:

- Clock Summary: Lists all clocks, their frequencies, and sources.
- Slack Analysis: Shows the difference between required and actual timing.
- Worst Path Analysis: Highlights the most critical timing paths.
- Violations: Identifies any setup or hold violations.

What to Look For:

- Positive slack values (indicate timing is met).
- No setup or hold violations.
- Correct clock propagation and relationships.

Utilization Reports

These reports show how much of the FPGA's resources are used.

Key Metrics:

- LUTs and Flip-Flops: Logic resource usage.
- Block RAMs and DSPs: Specialized resource usage.
- I/O Pins: Pin assignment and usage.

What to Look For:

- Resource usage within device limits.
- Balanced utilization across logic and memory.
- No over-utilization warnings.

Constraint Reports

There are various reports verify that constraints are correctly applied and respected. See the Viewing Logs and Reports section in the Radiant Help.

Key Checks:

- Pin Assignments: Ensure all I/O constraints are valid and conflict-free.
- Clock Constraints: Confirm all clocks are defined and used properly.
- False Paths and Exceptions: Validate that exceptions are correctly interpreted.

What to Look For:

- No invalid or ignored constraints.
- Warnings about unrecognized or conflicting constraints.
- Confirmation that generated clocks and asynchronous paths are handled correctly.

Power and Thermal Reports (if enabled)

These provide estimates of power consumption and thermal behavior.

What to Look For:

- Power within acceptable limits for your application.
- No thermal violations or hotspots.

Best Practices

- Review reports after every major design change.
- Use reports to guide constraint refinement.
- Cross-check timing and utilization to balance performance and resource use.
- Document findings and decisions for traceability.

Debugging Constraint Errors and Warnings

Constraint errors and warnings in Radiant indicate issues with how constraints are defined, interpreted, or applied during synthesis, placement, and routing. Properly diagnosing and resolving these messages is essential for achieving a successful and optimized FPGA implementation.

Where to Find Errors and Warnings

- Messages Window: Displays real-time feedback during synthesis and implementation.
- Constraint Validation Report: Summarizes all applied, ignored, or invalid constraints.
- Log Files: Provide detailed context for debugging complex issues.

Improperly defined or misapplied constraints can lead to errors, warnings, or suboptimal performance. Below are some of the most common issues and how to address them.

Common Types of Constraint Issues

Constraints Issues

Type	Description
Syntax Errors	Mistyped commands or incorrect formatting in constraint files.
Unrecognized Objects	Refers to signals, clocks, or pins that do not exist in the design.
Conflicting Constraints	Two or more constraints that contradict each other (e.g., overlapping pin assignments).
Ignored Constraints	Valid syntax, but the constraint is not applied due to tool limitations or context.

Unrecognized or Invalid Constraint Objects

Constraints refer to signals, clocks, or pins that do not exist in the design hierarchy.

Causes:

- Typographical errors in object names.
- Changes in design hierarchy not reflected in constraint files.

Solution:

- Use the Hierarchy Browser or Netlist Viewer to verify object names.
- Revalidate constraints after design changes.

Conflicting Pin Assignments

Multiple constraints assign different signals to the same pin or assign a signal to an unavailable pin.

Causes:

- Overlapping constraint files.
- Manual edits conflicting with GUI-based assignments.

Solution:

- Use the Device Constraint Editor to visualize and resolve conflicts.
- Consolidate constraints into a single source of truth.

Missing Clock Definitions

Timing analysis fails due to undefined clocks or incomplete clock propagation.

Causes:

- Clock signals not constrained in .sdc files.
- Generated clocks not declared.

Solution:

- Use create_clock and create_generated_clock commands.
- Review the Clock Summary in timing reports.

Incorrect Timing Exceptions

False paths or asynchronous paths are incorrectly defined, leading to missed violations or unnecessary exclusions.

Cause:

- Overuse or misapplication of set_false_path or set_clock_groups.

Solution:

- Validate all timing exceptions.
- Use comments to document why exceptions are applied.

Ignored Constraints

Constraints are syntactically correct but not applied during implementation.

Causes:

- Constraints applied to inactive or optimized-out logic.
- Tool limitations or unsupported syntax.

Solution:

- Check the Constraint Validation Report.
- Ensure constraints target active, synthesized objects.

Overly Aggressive Constraints

Constraints that are too tight can lead to routing congestion or failed timing closure.

Causes:

- Unrealistic timing requirements.
- Over-constraining placement regions.

Solution:

- Review timing slack and adjust constraints accordingly.
- Use floorplanning tools to balance placement.

Best Practices

- Validate constraints early and often.
- Use GUI tools to complement text-based editing.
- Document constraint intent for maintainability.
- Iterate based on report feedback.

Constraint Editor Limitations and Workarounds

While Radiant provides powerful graphical and text-based tools for managing constraints, users may encounter limitations in the Constraint Editors—particularly the Device Constraint Editor and Physical Designer. Understanding these limitations and applying effective workarounds can help maintain design accuracy and efficiency.

Limited Support for Complex Timing Constraints

The graphical editors may not support advanced timing constraints such as multi-cycle paths, false paths, or asynchronous clock relationships.

Workaround:

- Use .sdc files for advanced timing constraints.

No Real-Time Cross-Referencing with RTL

Constraint editors do not dynamically link to RTL code, making it harder to verify signal names and hierarchy.

Workaround:

- Use the Hierarchy Browser or Netlist Viewer to confirm object names.
- Export signal lists from synthesis reports to assist with constraint entry.

Limited Visualization of Timing Paths

The editors focus on physical placement and pin assignments but do not visualize timing paths or slack directly.

Workaround:

- Use the Timing Report Viewer post-implementation to analyze critical paths.
- Combine with Physical Designer to manually correlate timing paths with placement.

Constraint Conflicts Not Always Highlighted

Conflicting constraints (e.g., overlapping pin assignments or region constraints) may not be flagged until synthesis or implementation.

Workaround:

- Regularly run Constraint Validation before synthesis.
- Use the Messages Window and Constraint Validation Report to catch issues early.

Limited Batch Editing Capabilities

Editing multiple constraints (e.g., pin assignments) in bulk is cumbersome in the GUI.

Workaround:

- Export constraints to .csv format.
- Edit in a spreadsheet or text editor, then re-import into Radiant.

No Support for Conditional Constraints

Constraint editors do not support conditional logic (e.g., constraints based on configuration modes).

Workaround:

- Use set_case_analysis in .sdc files to simulate conditional behavior.
- Maintain separate constraint files for different configurations.

Best Practices

- Combine GUI and script-based workflows for flexibility.
- Validate constraints regularly using built-in reports.
- Document constraint intent to aid debugging and collaboration.

Debugging Workflow

1. Identify the Message
   - Read the full error or warning message.
   - Note the file, line number, and object involved.
2. Check for Typos or Syntax Issues
   - Use the correct syntax.
   - Refer to the Radiant constraint reference guide or documentation.
3. Verify Object Names
   - Ensure that signal, clock, or pin names match those in the synthesized netlist.
   - Use the Netlist Analyzer to confirm names.
4. Resolve Conflicts
   - Review overlapping or contradictory constraints.
   - Use the Device Constraint Editor or Physical Designer to visualize assignments.
5. Re-run Validation
   - After making corrections, re-run synthesis or constraint validation to confirm resolution.

Tips for Effective Debugging

- Use Comments: Annotate constraint files to explain intent.
- Modularize Constraints: Separate timing and physical constraints into different files for clarity.
- Incremental Testing: Apply constraints in stages to isolate issues.
- Use GUI Tools: Editors like the Device Constraint Editor help catch errors visually.

References and Further Readings

Radiant Software Help

The Lattice Radiant Help includes detailed information on constraints in the following topics:

- User Guides > Applying Design Constraints
- Reference Guides > Constraints Reference Guide

  A comprehensive guide to library element HDL attributes is available in the FPGA Libraries Reference Guide. For specific applications, refer to the technical notes that are available on the Lattice web site. Editing these library element attributes is not recommended, as they are usually generated from an array of choices that are made for module generation using IP Catalog.

Documents

- Lattice Radiant Software Constraints Propagation Engine (FPGA-AN-02097)
- Timing Constraints Methodology for Source-Synchronous Interfaces (FPGA-AN-02078)
- Lattice Radiant Timing Constraints Methodology (FPGA-AN-02059)
- Lattice Radiant Software <*version*> Help in PDF format

Training Course/Video Tutorial

- Timing Constraints Deep Dive with Radiant

  This course is available at the Lattice Insights training portal. It is a comprehensive intermediate-level course designed to help FPGA developers master timing constraints using Lattice's Radiant software.

  To access the full course content including videos, labs, and downloadable materials, sign up or log in to the Lattice Insights portal.
- Radiant Video Series 4.2: Creating Timing Constraints

  Part of the Lattice Radiant Design Software Video Series, this video shows you how to use the Timing Constraint Editor to create timing constraints for synthesis.

Technical Support Database

- Lattice Answer Database

  The Lattice Answer Database is an official technical support resource provided by Lattice Semiconductor. It contains technical solutions, frequently asked questions (FAQs), discussions of known issues, and design tips and troubleshooting guidance.

Revision History

The following table gives the revision history for this document.

Date	Radiant Version	Description
06/26/2026	2025.1	Updates for 2026.1 release.
12/11/2025	2025.2	Initial Release.